

Mc  
Graw  
Hill

计 算 机 科 学 丛 书

原书第5版

# 计算机组成

(加) Carl Hamacher Zvonko Vranesic Safwat Zaky 著 张红光 张健民 李莹 等译

INTERNATIONAL EDITION

## COMPUTER ORGANIZATION

fifth edition



Carl Hamacher  
Zvonko Vranesic  
Safwat Zaky

Computer Organization  
Fifth Edition



机械工业出版社  
China Machine Press



这本经典教材的第5版对计算机组成结构进行了全面的概括。它介绍了硬件设计的原理，并且说明了硬件设计是如何受软件需求影响的。本书素材经过反复改写，反映了计算机技术发展的现状。例如使用了典型的商用处理器来说明一般的概念，并用ARM、68000以及Pentium处理器作为主要的结构范例。书中还包括了有关嵌入式系统的讨论。本书涉及现代计算机设计的各个方面——处理器、输入/输出、存储器、外围设备以及通信链接，重点放在完整的计算机系统设计上。

本书结构清晰，使用灵活，主要面向已经学习了逻辑电路课程的学生，书后附有逻辑电路的内容介绍，没有这方面基础知识的学生也可使用。

作者简介

**Carl Hamacher** 加拿大滑铁卢大学工程物理学士，加拿大女皇大学电子工程硕士，美国Syracuse大学电子工程博士。自1991年起任女皇大学电子及计算机工程学院教授。他的研究方向为多处理器与多计算机，侧重于网络互连。

**Zvonko Vranesic** 先后获得加拿大多伦多大学电子工程学士、硕士及博士学位。他现在的研究方向包括：计算机体系结构、可编程VLSI技术及多值逻辑系统。

**Safwat Zaky** 埃及开罗大学电子工程和数学双学士，后获多伦多大学电子工程硕士及博士学位。他的研究方向为计算机体系结构、数字电路可靠性及电磁相容性分析。他还是《Microcomputer Structures》一书的合著者。

ISBN 7-111-14262-4



9 787111 142621



华章图书

华章网站 <http://www.hzbook.com>

网上购书: [www.china-pub.com](http://www.china-pub.com)

北京市西城区百万庄南街1号 100037

读者服务热线: (010)68995259, 68995264

读者服务信箱: [hzedu@hzbook.com](mailto:hzedu@hzbook.com)

ISBN 7-111-14262-4/TP · 3535

定价: 59.00 元

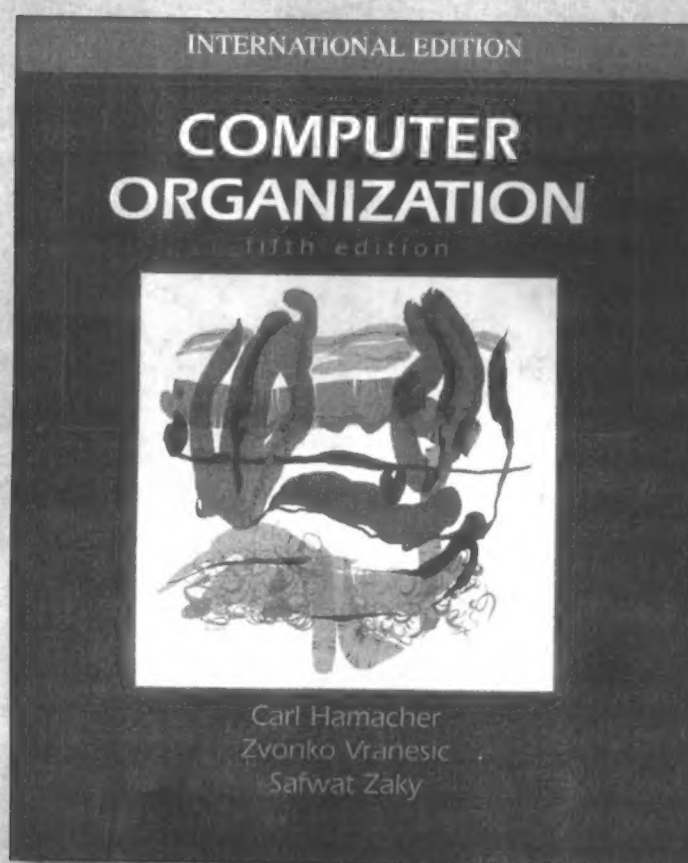


计 算 机 科 学 丛 书

原书第5版

# 计算机组成

(加) Carl Hamacher Zvonko Vranesic Safwat Zaky 著 张红光 张健民 李莹 等译



**Computer Organization**  
Fifth Edition



机械工业出版社  
China Machine Press

本书是计算机组成的入门级教程,全面地介绍了计算机组成结构、操作、性能的基本概念,还介绍了有关外围设备、处理器系列模型以及嵌入式系统的一些主要内容。书中知识具有很强的实用性,并涵盖了当今许多先进的技术和设计思想。

本书知识结构相对独立,需要读者具备计算机高级语言程序设计和数字逻辑电路的基本知识。本书适合用作高等院校电子工程、计算机工程和计算机专业计算机组成课程的教材。

Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, Fifth Edition (ISBN 0-07-232086-9).

Copyright © 2002 by The McGraw-Hill Companies, Inc.

Original English edition published by The McGraw-Hill Companies, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Simplified Chinese translation edition jointly published by McGraw-Hill Education (Asia) Co. and China Machine Press.

本书中文简体字翻译版由机械工业出版社和美国麦格劳-希尔教育(亚洲)出版公司合作出版。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有McGraw-Hill公司防伪标签,无标签者不得销售。

版权所有,侵权必究。

**本书版权登记号: 图字: 01-2002-2183**

### **图书在版编目(CIP)数据**

计算机组成(原书第5版)/(加)哈马彻(Hamacher, C.)等著;张红光等译.-北京:机械工业出版社,2004.7

(计算机科学丛书)

书名原文: Computer Organization, Fifth Edition

ISBN 7-111-14262-4

I. 计… II. ①哈… ②张… III. 计算机体系结构-教材 IV. TP303

中国版本图书馆CIP数据核字(2004)第027656号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑: 刘 渊

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2004年7月第1版第1次印刷

787mm×1092mm 1/16·37.5印张

印数: 0 001-4 000册

定价: 59.00元

凡购本书,如有倒页、脱页、缺页,由本社发行部调换  
本社购书热线:(010) 68326294



# 出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall, Addison-Wesley, McGraw-Hill, Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum, Stroustrup, Kernighan, Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及度藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T., Stanford, U.C. Berkeley, C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程,而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下,读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑,这些因素使我们的图书有了质量的保证,但我们的目标是尽善尽美,而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正,我们的联系方式如下:

电子邮件: [hzedu@hzbook.com](mailto:hzedu@hzbook.com)

联系电话: (010) 68995264

联系地址: 北京市西城区百万庄南街1号

邮政编码: 100037



# 专家指导委员会

(按姓氏笔画顺序)

尤晋元  
石教英  
张立昂  
邵维忠  
周立柱  
范明  
袁崇义  
谢希仁

王珊  
吕建  
李伟琴  
陆丽娜  
周克定  
郑国梁  
高传善  
裘宗燕

冯博琴  
孙玉芳  
李师贤  
陆鑫达  
周傲英  
施伯乐  
梅宏  
戴葵

史忠植  
吴世忠  
李建中  
陈向群  
孟小峰  
钟玉琢  
程旭

史美林  
吴时霖  
杨冬青  
周伯生  
岳丽华  
唐世渭  
程时端

## 秘 书 组

武卫东

温莉芳

刘江

杨海玲

# 译者序

计算机组成原理是学习计算机工程与计算机专业基础知识的基础，计算机系统设计的复杂性决定了本书论述内容的综合性和广泛性。计算机系统设计包含多个领域知识的研究与应用，而计算机体系结构的建立包括设计算法的选择、设计方案的确立、系统成本与系统性能的权衡以及硬件功能与软件功能的权衡等方面，需要考证大量的实验数据并积累实践经验。学习计算机组成原理就是要学会从多层面去理解所要解决的具体问题，从表面的现象中挖掘出系统内在的、深层次的联系，从硬件和软件两个角度探讨最佳的解决途径和解决方案。

《计算机组成》(*Computer Organization*)是一本经典的电子工程与计算机专业本科教科书，它的第1版于1978年问世，之后陆续出版了第2~5版，我们本次翻译的第5版目前已成为多所世界知名大学的本科教材。本书知识结构合理，知识点全面完整，基本概念广泛而新颖。更可贵的是书中以流行的商用处理器作为范例，描述了各种基本知识和基本概念的应用方法和应用过程，使书中知识具有很强的实用性，同时还涵盖了当今许多先进的技术和设计思想。

本书由12章和5个附录部分组成，全面地介绍了计算机组成、计算机操作、计算机性能的基本概念，还介绍了外围设备、处理器系列模型以及嵌入式系统的一些主要内容。书中知识独立，适合讲授或自学。附录中的内容是对正文的补充，将两者结合起来学习可以收到良好的效果。

本书由张红光组织并翻译，张健民、李莹、蒋跃军参与了大量的翻译和校对工作。参加本书翻译、校对及整理工作的还有张楠、王华、徐巧丽、房金花等。南开大学微电子中心的李福才副教授对本书的翻译工作给予了大量的帮助和指导，在此一并表示感谢。

由于译者水平有限，译文中疏漏和错误在所难免，敬请读者批评指正。

译者

2003年12月于南开园



## 作者简介

**Carl Hamacher** 加拿大滑铁卢大学工程物理学士，加拿大金斯顿女皇大学（Queen's University at Kingston）电子工程硕士，纽约州Syracuse大学电子工程博士。1968~1990年任多伦多大学电子工程及计算机科学学院教授，1984~1988年担任计算机系统研究所所长，1988~1990年担任工程科学部主席。自1991年1月起任女皇大学电子及计算机工程学院教授，1991~1996年任应用科学系主任。1978~1979年曾作为访问学者到加利福尼亚州San Jose的IBM研究实验室工作。1986年作为访问研究员在法国Grenoble大学电路与系统实验室工作。1996~1997年作为访问教授在加州大学Riverside分校计算机科学系和法国Paris VI大学LIP6实验室工作。

他的研究方向为多处理器与多计算机，侧重于网络互连。

**Zvonko Vranesic** 先后获得加拿大多伦多大学电子工程学士、硕士及博士学位。1963~1965年在加拿大安大略省Bramalea的北方电子有限公司（the Northern Electric Co., Ltd.）任设计工程师。1968年进入多伦多大学工作，现为电子及计算机工程学院和计算机科学学院教授。1978~1979年作为高级访问学者访问英国剑桥大学，1984~1985年访问法国Paris VI大学。2000~2001年担任多伦多Altera公司高级软件工程师。1995~2000年任多伦多大学工程科学学院客座教授。

他现在的研究方向包括：计算机体系结构、可编程VLSI技术及多值逻辑系统。他也是另外三本书——《Fundamentals of Digital Logic with VHDL Design》、《Microcomputer Structures》及《Field-Programmable Gate Arrays》的合著者。1990年由于其“对本科实验指导的创新和与众不同的贡献”获Wighton奖金。

**Safwat Zaky** 埃及开罗大学电子工程和数学双学士，后获多伦多大学电子工程硕士及博士学位。1969~1972年在加拿大安大略省的贝尔北方研究室（Bell Northern Research）从事在大规模存储与电话交换中应用光电子及磁性的研究。1973年进入多伦多大学工作，现为电子及计算机工程学院和计算机科学学院教授，并且是电子与计算机工程学院院长。1980~1981年他曾作为高级访问学者在英国剑桥大学计算机实验室工作。

他的研究方向为计算机体系结构、数字电路可靠性及电磁相容性分析。他还是《Microcomputer Structures》一书的合著者，并且荣获了IEEE“第三千禧奖章”（the IEEE Third Millennium Medal）。

# 前 言

本书适用于电子工程、计算机工程和计算机专业有关计算机组成方面的初级课程。本书的知识结构是相对独立的,假设读者已有了计算机高级语言程序设计的基本知识。

许多学习计算机组成的学生都已经学习了数字逻辑电路这一门引导课程。因此,本书的主体内容没有包含这一部分的知识。但是我们为有需要的读者提供了逻辑电路方面的详尽附录。

本书融入了作者在为电子及计算机工程、计算机科学与工程专业的本科生讲授计算机组成原理时所积累的丰富经验。在这个领域我们总是采用从实际出发的观点进行教学,因此形成了本书在内容上的一个关键考虑,即使用从商用计算机中提取的例子来说明计算机的组成原理。本书中的主要例子来源于以下处理器:ARM、Motorola 680X0、Intel Pentium 及Sun UltraSPARC。

读者必须清楚地认识到,数字系统的设计并不是应用最佳设计算法的简单过程。许多设计决策取决于大量试探性的判断和经验。这包括在一系列选择方案中进行成本/性能、硬件/软件的权衡。我们的目标就是把这些思想传达给读者。

我们努力提供足够多的细节,鼓励学生在处理那些看上去很明显的问题时进行深层次的挖掘,我们相信最佳的途径是使用那些已被充分验证过的真实例子。框图是描述计算机组成特征的有效方法,但是它们容易使问题过于简单化。因此,必须使用各种实现方案的细节作为补充。

本书可以作为工程学或计算机专业一个学期的课程用书。它对于软件和硬件方向的学生均适用。尽管本书侧重于硬件,我们仍阐述了大量软件方面的问题,包括与指令执行性能、系统级并行操作协调以及实时应用程序相关的编译器与操作系统的基本知识。计算机专业人员有必要了解软件与硬件之间的交互与权衡问题。

## 本书的内容

下面我们按章节顺序来介绍一下本书的内容。前8章涵盖了计算机组成、操作及性能的基本概念,后4章讨论了嵌入式系统、外围设备、处理器系列的演变模型以及大型计算机系统。

第1章对计算机硬件和软件给出了总体的描述,并对在后续章节中将会深入研究的术语进行了概括性介绍。该章介绍了基本功能部件以及将它们相互连接组成一个完整计算机系统的方法,还介绍了系统软件的作用,并讨论了性能评估的基本内容。还介绍了计算机的发展简史。

第2章系统地介绍了机器指令、寻址技术和指令序列。为了便于讨论有效地址的生成,我们引入了二进制补码运算。用于讨论循环、子程序、简单的输入输出编程、排序和链表操作的程序示例均在机器指令级别上使用通用的汇编语言表示。

第3章以三种商用处理器——ARM、68000和Pentium为例说明了第2章中概念的具体实现。其中ARM处理器诠释了RISC设计风格,68000使用了易于接受的CISC设计,而Pentium处理器综合了RISC和CISC的设计风格并成为最成功的商业设计。这些内容构成三个独立而完整的部分。每个部分都包含第2章中提出的所有例子,它们使用指定的处理器进行了实现。学习其中任何一部分都能够满足继续学习本书后面章节的需要。如果实验室使用了书中介绍的其中一种处理器,那么第3章中的相关部分就可以与第2章同时讲授。



输入输出结构将在第4章中介绍。这一章讲述了I/O数据传输同步的基本知识以及一系列渐趋复杂的I/O结构。我们对中断和直接存储器访问方法进行了详细的描述,包括对操作系统中软件中断作用的具体讨论。还以PCI、SCSI以及USB标准为代表介绍了总线协议与标准。

第5章讨论了半导体存储器,包括SDRAM、Rambus和闪存(flash memory)的实现。作为增加主存储器带宽的方法,本章还介绍了高速缓存(cache)和多模块存储器系统。其中对高速缓存进行了详细讨论,包括性能建模。还介绍了虚拟存储器系统、存储器管理和快速地址转换技术,并将磁盘和光盘作为存储器体系结构的一部分进行了讨论。

第6章介绍了计算机中的算术单元,对二进制补码数定点加、减、乘、除硬件的逻辑设计操作作了描述。这一章还解释了超前进位加法器和快速乘法器,并描述了Booth乘法器重编码和进位保留加法技术。另外本章还介绍了IEEE标准中浮点数的表示与操作。

第7章在寄存器传送层次上介绍了处理器取指与执行的实现,接下来讨论了使用硬布线和微程序控制方式实现的处理器。

第8章详细介绍了流水线和多功能单元在高性能处理器设计中的使用。这一章还探讨了编译器的作用以及流水线执行与指令集设计之间的关系。本章还对超标量处理器进行了讨论,并以Sun公司的UltraSPARC II处理器为例阐述了相关概念。

今天越来越多的处理器被用于嵌入式系统而不是通用计算机中。大量低成本的应用都需要将处理I/O和定时功能集成在单个芯片上,这一日趋重要的问题将在第9章介绍。在第9章中还将讨论系统集成、互连及实时软件设计问题。

第10章讲述外围设备和计算机互连的有关知识,介绍了典型的输入/输出设备及支持计算机图形应用所需要的硬件设备,还讨论了DSL等一般通信链路。

第11章介绍了ARM、Motorola和Intel处理器系列的演变过程。这一章强调的是可获得更高性能要求的设计改变,还讨论了PowerPC、SPARC、Alpha和Intel IA-64系列。

第12章将计算机组成的讨论扩展到多处理器、并行操作的大型系统层面上。这一章描述了对多处理器的互连网络,还对高速缓存的一致性控制、共享存储器与消息传递策略作了介绍。

## 第5版的特点

本书的第5版对内容和结构安排主要作了以下几个方面的改动:

- 第4版中的第2章在第5版中拆为两章,即第2章与第3章。本版中第2章对基础问题作了扩展处理,并用一般的指令进行解释,还提供了很多解决典型问题的程序实例,包括数字的与非数字的。而第3章则使用了ARM、68000及Pentium处理器等指令集来说明如何以RISC和CISC设计风格实现指令集设计的基本概念。
- 对流水线和多功能单元在处理器设计中作用的讨论进行了很大程度的扩展,把UltraSPARC体系结构作为增强设计性能的一个特殊例子作了介绍。
- 加入了嵌入式处理器系统这一新内容,并以典型系统的通用设计作为详细讨论应用举例的基础。

除了上述的主要变化外,本版的多个章节还加入了当今许多先进的技术和设计方法。

## 课程进度安排

本书适合作为大学或学院计算机组成方面入门课程的教材,总学时为一学期。

书中提供了多于一个学期课程所要讲授的内容，其核心部分是第1章至第8章。如果学生还未学习过逻辑电路，则应在第4章之前学习附录A中的基本内容。

第9章至第12章包含了大量实用的内容，教师可根据实际情况自主安排所要讲授的内容，特别是第9章有关嵌入式系统的讨论和第10章关于大多数个人计算机中所使用的硬件设备描述。

## 致谢

在此，我们向许多在第5版出版的准备期间提供帮助的朋友表示衷心的感谢。Gail Burgess和Kelly Chan协助我们完成了原稿的技术处理。Alex Grbic、Frank Hsu 和Robert Lu向我们提供了大量有价值的程序实例。我们的同事 Tarek Abdelrahman、Stephen Brown、Paul Chow、Glenn Gulak 和 Jonathan Rose提出了许多建设性的建议。我们特别要感谢 Stephen和Tarek在重要细节问题上提供的帮助。俄亥俄州立大学的Gojko Babic，弗吉尼亚工业学院和州立大学的Nathaniel Davis，普度大学的Jose Fortes，赖斯大学的John Greiner，旧金山州立大学的Sung Hu，宾州州立大学的Ali Hurson，得州大学奥斯汀分校的 Lizy Kurian John，德国弗赖堡Albert Ludwigs大学的Stefan Leue，Northeastern大学的Fabrizio Lombardi，George Mason 大学的 Daniel Tabak，Rensselaer Polytechnic 学院的 John Valois也给予我们很多出色的建议。还要感谢 Eli Vranesic允许我们使用他的作品“Fall in High Park”作为本书的封面，他用计算机完成了这幅作品。最后，我们衷心地感谢编辑 Catherine Fields Shultz及她在 McGraw-Hill的同事们的支持，他们是 Kelley Butcher、Michelle Flomenhoft、Kalah Graham、Betsy Jones、Rick Noel、Heather Sabo 和Christine Walker。

Carl Hamacher  
Zvonko Vranesic  
Safwat Zaky



# 目 录

出版者的话	
专家指导委员会	
译者序	
作者简介	
前言	
第1章 计算机的基本结构	1
1.1 计算机的类型	1
1.2 功能部件	2
1.2.1 输入设备	3
1.2.2 存储器	3
1.2.3 运算器	4
1.2.4 输出设备	4
1.2.5 控制器	4
1.3 基本操作概念	5
1.4 总线结构	6
1.5 软件	7
1.6 性能	9
1.6.1 处理器时钟	10
1.6.2 基本性能公式	10
1.6.3 流水线和超标量操作	10
1.6.4 时钟频率	11
1.6.5 指令集: CISC和RISC	11
1.6.6 编译器	12
1.6.7 性能测量	12
1.7 多处理器和多计算机	13
1.8 发展历程	13
1.8.1 第一代计算机	13
1.8.2 第二代计算机	14
1.8.3 第三代计算机	14
1.8.4 第四代计算机	14
1.8.5 后第四代计算机	14
1.8.6 性能的发展	14
1.9 结束语	15
习题	15
参考文献	16

第2章 机器指令和程序	17
2.1 数、算术运算以及字符	18
2.1.1 数的表示	18
2.1.2 正数的加法	19
2.1.3 有符号数的加法和减法	19
2.1.4 整数算术运算中的溢出	21
2.1.5 字符	22
2.2 内存单元和寻址	22
2.2.1 按字节寻址能力	23
2.2.2 big-endian和little-endian分配	23
2.2.3 字的对齐	24
2.2.4 访问数、字符和字符串	24
2.3 存储器操作	24
2.4 指令和指令序列	25
2.4.1 寄存器传送标记	25
2.4.2 汇编语言符号	25
2.4.3 基本指令类型	26
2.4.4 指令执行和线性序列	29
2.4.5 转移	30
2.4.6 条件码	31
2.4.7 生成存储器地址	32
2.5 寻址方式	32
2.5.1 变量和常数的实现	33
2.5.2 间接和指针	34
2.5.3 变址和数组	36
2.5.4 相对寻址	38
2.5.5 附加方式	39
2.6 汇编语言	40
2.6.1 汇编指示	41
2.6.2 程序的汇编和执行	43
2.6.3 数的表示	44
2.7 基本输入/输出操作	44
2.8 堆栈和队列	47
2.9 子程序	50
2.9.1 子程序嵌套及处理器堆栈	51

2.9.2 参数传递 .....	51	3.12 I/O操作 .....	102
2.9.3 堆栈的结构 .....	52	3.13 堆栈和子程序 .....	103
2.10 附加的指令 .....	56	3.14 逻辑指令 .....	107
2.10.1 逻辑指令 .....	56	3.15 实例程序 .....	108
2.10.2 移位和循环移位指令 .....	57	3.15.1 向量点积程序 .....	108
2.10.3 乘法和除法 .....	60	3.15.2 字节排序程序 .....	108
2.11 实例程序 .....	60	3.15.3 链表的插入和删除子程序 .....	109
2.11.1 向量点积程序 .....	60	部分III IA-32 Pentium 实例 .....	109
2.11.2 字节排序程序 .....	61	3.16 寄存器和寻址方式 .....	110
2.11.3 链表 .....	62	3.16.1 IA-32寄存器结构 .....	111
2.12 机器指令的编码 .....	66	3.16.2 IA-32寻址方式 .....	113
2.13 结束语 .....	69	3.17 IA-32指令 .....	116
习题 .....	69	3.18 IA-32汇编语言 .....	121
第3章 ARM、Motorola与Intel指令集 .....	73	3.19 程序流控制 .....	122
部分I ARM实例 .....	73	3.19.1 条件跳转及条件码标志 .....	122
3.1 寄存器、内存访问及数据传递 .....	73	3.19.2 无条件跳转 .....	123
3.1.1 寄存器结构 .....	74	3.20 逻辑和移位/循环移位指令 .....	123
3.1.2 内存访问指令和寻址方式 .....	74	3.20.1 逻辑操作 .....	123
3.1.3 寄存器传送指令 .....	80	3.20.2 移位与循环移位操作 .....	123
3.2 算术和逻辑指令 .....	80	3.21 I/O操作 .....	124
3.2.1 算术指令 .....	80	3.21.1 存储器映射I/O .....	124
3.2.2 逻辑指令 .....	81	3.21.2 独立I/O .....	124
3.3 转移指令 .....	82	3.21.3 块传送 .....	125
3.3.1 设置条件码 .....	83	3.22 子程序 .....	126
3.3.2 用于数值相加的循环程序 .....	83	3.23 其他指令 .....	127
3.4 汇编语言 .....	84	3.23.1 乘法与除法指令 .....	128
3.5 I/O操作 .....	85	3.23.2 多媒体扩展 (MMX) 指令 .....	130
3.6 子程序 .....	86	3.23.3 向量 (SIMD) 指令 .....	131
3.7 实例程序 .....	89	3.24 实例程序 .....	131
3.7.1 向量点积程序 .....	89	3.24.1 向量点积程序 .....	131
3.7.2 字节排序程序 .....	90	3.24.2 字节排序程序 .....	131
3.7.3 链表的插入和删除子程序 .....	91	3.24.3 链表的插入与删除子程序 .....	132
部分II 68000实例 .....	93	3.25 结束语 .....	133
3.8 寄存器与寻址方式 .....	93	习题 .....	134
3.8.1 68000寄存器结构 .....	93	参考文献 .....	143
3.8.2 寻址方式 .....	93	第4章 输入输出组织结构 .....	145
3.9 指令 .....	97	4.1 访问I/O设备 .....	145
3.10 汇编语言 .....	99	4.2 中断 .....	148
3.11 程序流控制 .....	100	4.2.1 中断的硬件 .....	149
3.11.1 条件码标志 .....	100	4.2.2 中断的允许和禁止 .....	150
3.11.2 转移指令 .....	100	4.2.3 处理多台设备 .....	151

4.2.4 控制设备请求 .....	154	5.5.1 映射功能 .....	224
4.2.5 异常 .....	155	5.5.2 替换算法 .....	227
4.2.6 在操作系统中使用的中断 .....	156	5.5.3 映射技术的例子 .....	228
4.3 处理器举例 .....	159	5.5.4 商用处理器中高速缓存的例子 .....	230
4.3.1 ARM中断结构 .....	159	5.6 性能因素 .....	233
4.3.2 68000中断结构 .....	163	5.6.1 交叉 .....	233
4.3.3 Pentium的中断结构 .....	164	5.6.2 命中率和失效开销 .....	235
4.4 直接存储器访问 .....	166	5.6.3 处理器芯片上的高速缓存 .....	237
4.5 总线 .....	170	5.6.4 其他改进 .....	237
4.5.1 同步总线 .....	171	5.7 虚拟存储器 .....	239
4.5.2 异步总线 .....	173	5.8 存储器管理需求 .....	242
4.5.3 讨论 .....	175	5.9 辅助存储器 .....	243
4.6 接口电路 .....	176	5.9.1 磁性硬盘 .....	243
4.6.1 并行端口 .....	176	5.9.2 光盘 .....	249
4.6.2 串行端口 .....	182	5.9.3 磁带系统 .....	253
4.7 标准I/O接口 .....	184	5.10 结束语 .....	254
4.7.1 外围部件互连 (PCI) 总线 .....	185	习题 .....	255
4.7.2 SCSI总线 .....	189	参考文献 .....	259
4.7.3 通用串行总线 (USB) .....	193	第6章 算术运算 .....	261
4.8 结束语 .....	200	6.1 有符号数加减法 .....	261
习题 .....	200	6.2 快速加法器设计 .....	264
参考文献 .....	205	6.3 正数乘法 .....	267
第5章 存储器系统 .....	207	6.4 有符号操作数乘法 .....	270
5.1 基本概念 .....	207	6.5 快速乘法 .....	272
5.2 半导体随机存储器 .....	209	6.5.1 乘数位偶重编码 .....	272
5.2.1 存储器芯片的内部组织结构 .....	210	6.5.2 求和项的进位保留加法 .....	273
5.2.2 静态存储器 .....	211	6.6 整数除法 .....	276
5.2.3 异步动态随机存储器 .....	212	6.7 浮点数及其操作 .....	278
5.2.4 同步动态随机存储器 .....	214	6.7.1 浮点数的IEEE标准 .....	279
5.2.5 大容量存储器的结构 .....	216	6.7.2 浮点数算术运算 .....	281
5.2.6 存储器系统因素 .....	217	6.7.3 保护位与截取 .....	282
5.2.7 Rambus存储器 .....	218	6.7.4 浮点操作的实现 .....	283
5.3 只读存储器 .....	219	6.8 结束语 .....	285
5.3.1 ROM .....	220	习题 .....	285
5.3.2 PROM .....	220	参考文献 .....	290
5.3.3 EPROM .....	220	第7章 基本处理部件 .....	291
5.3.4 EEPROM .....	221	7.1 一些基本概念 .....	291
5.3.5 闪存 .....	221	7.1.1 寄存器传送 .....	293
5.4 速度、容量和成本 .....	222	7.1.2 执行算术或逻辑操作 .....	294
5.5 高速缓存 .....	223	7.1.3 从存储器中取出一个字 .....	295
		7.1.4 向存储器中存储一个字 .....	296

7.2 一条完整指令的执行 .....	297	参考文献 .....	358
7.3 多总线结构 .....	298	第9章 嵌入式系统 .....	359
7.4 硬件控制 .....	300	9.1 嵌入式系统的实例 .....	359
7.5 微程序控制 .....	302	9.1.1 微波炉 .....	360
7.5.1 微指令 .....	304	9.1.2 数码照相机 .....	361
7.5.2 微程序的顺序 .....	306	9.1.3 家用遥测技术 .....	362
7.5.3 宽转移寻址方式 .....	307	9.2 嵌入式应用中的处理器芯片 .....	363
7.5.4 带有下一地址字段的微指令 .....	309	9.3 一个简单的微控制器 .....	364
7.5.5 预取微指令 .....	311	9.3.1 并行I/O端口 .....	364
7.5.6 仿真 .....	313	9.3.2 串行I/O接口 .....	366
7.6 结束语 .....	313	9.3.3 计数器/定时器 .....	367
习题 .....	314	9.3.4 中断控制机制 .....	368
第8章 流水线 .....	319	9.4 程序设计问题 .....	369
8.1 基本概念 .....	319	9.4.1 轮询方法 .....	369
8.1.1 高速缓存的作用 .....	321	9.4.2 中断方法 .....	372
8.1.2 流水线性能 .....	322	9.5 I/O设备的时序限制 .....	373
8.2 数据阻塞 .....	324	9.5.1 通过环形缓冲区做传送的C程序 .....	374
8.2.1 操作数传递 .....	325	9.5.2 通过环形缓冲区做传送的汇编语言 程序 .....	374
8.2.2 用软件方法处理数据阻塞 .....	326	9.6 反应计时器实例 .....	374
8.2.3 副作用 .....	327	9.6.1 用于反应计时器的C程序 .....	377
8.3 指令阻塞 .....	327	9.6.2 用于反应计时器的汇编语言程序 .....	379
8.3.1 无条件转移 .....	327	9.6.3 最后评价 .....	380
8.3.2 条件转移和转移预测 .....	330	9.7 嵌入式处理器系列 .....	380
8.4 对指令集的影响 .....	334	9.7.1 基于Intel 8051的微控制器 .....	381
8.4.1 寻址方式 .....	334	9.7.2 Motorola微控制器 .....	381
8.4.2 条件码 .....	336	9.7.3 ARM微控制器 .....	382
8.5 数据通路和控制 .....	336	9.8 设计问题 .....	382
8.6 超标量操作 .....	338	9.9 片上系统 .....	384
8.6.1 无序执行 .....	339	9.10 结束语 .....	386
8.6.2 执行完成 .....	340	习题 .....	387
8.6.3 调度操作 .....	341	参考文献 .....	389
8.7 UltraSPARC II实例 .....	342	第10章 计算机外围设备 .....	391
8.7.1 SPARC体系结构 .....	342	10.1 输入设备 .....	391
8.7.2 UltraSPARC II .....	346	10.1.1 键盘 .....	392
8.7.3 流水线结构 .....	346	10.1.2 鼠标 .....	392
8.8 性能考虑 .....	353	10.1.3 跟踪球、操作杆和触摸垫 .....	392
8.8.1 指令阻塞的影响 .....	354	10.1.4 扫描仪 .....	394
8.8.2 流水线的段数 .....	355	10.2 输出设备 .....	394
8.9 结束语 .....	355	10.2.1 视频显示器 .....	394
习题 .....	356		

10.2.2 平面显示器 .....	395	11.7.2 条件执行 .....	424
10.2.3 打印机 .....	396	11.7.3 推测性装入 .....	425
10.2.4 图形加速卡 .....	396	11.7.4 寄存器和寄存器堆栈 .....	425
10.3 串行通信连接 .....	398	11.7.5 Itanium 处理器 .....	427
10.3.1 异步传输 .....	400	11.8 堆栈处理器 .....	427
10.3.2 同步传输 .....	401	11.8.1 堆栈结构 .....	428
10.3.3 标准通信接口 .....	404	11.8.2 堆栈指令 .....	430
10.4 结束语 .....	405	11.8.3 堆栈中的硬件寄存器 .....	433
习题 .....	405	11.9 结束语 .....	433
第11章 处理器系列 .....	409	习题 .....	434
11.1 ARM系列 .....	410	参考文献 .....	435
11.1.1 Thumb指令集 .....	410	第12章 大型计算机系统 .....	437
11.1.2 处理器和CPU内核 .....	411	12.1 并行处理的形式 .....	438
11.2 Motorola 680X0和ColdFire系列 .....	412	12.2 阵列处理器 .....	439
11.2.1 68020处理器 .....	412	12.3 通用多处理器结构 .....	440
11.2.2 68030和68040处理器的改进 .....	414	12.4 互连网络 .....	441
11.2.3 68060处理器 .....	414	12.4.1 信号总线 .....	442
11.2.4 ColdFire系列 .....	414	12.4.2 纵横 (Crossbar) 网络 .....	443
11.3 Intel IA-32系列 .....	415	12.4.3 多段网络 .....	444
11.3.1 IA-32存储器分段 .....	415	12.4.4 超立方体网络 .....	445
11.3.2 16位模式 .....	416	12.4.5 网状网络 .....	446
11.3.3 80386和80486 处理器 .....	417	12.4.6 树状网络 .....	446
11.3.4 Pentium处理器 .....	417	12.4.7 环状网络 .....	447
11.3.5 Pentium Pro 处理器 .....	417	12.4.8 实用性因素 .....	448
11.3.6 Pentium II和Pentium III处理器 .....	418	12.4.9 混合拓扑网络 .....	450
11.3.7 Pentium 4处理器 .....	418	12.4.10 对称式多处理器 .....	450
11.3.8 AMD公司的IA-32处理器 .....	419	12.5 多处理器的存储器组织结构 .....	451
11.4 PowerPC 系列 .....	419	12.6 程序并行性与共享变量 .....	452
11.4.1 寄存器集 .....	419	12.6.1 共享变量访问 .....	452
11.4.2 存储器寻址方式 .....	419	12.6.2 高速缓存一致性 .....	454
11.4.3 指令 .....	419	12.6.3 加锁和高速缓存一致性 .....	456
11.4.4 PowerPC处理器 .....	420	12.7 多计算机 .....	456
11.5 Sun公司SPARC系列 .....	421	12.7.1 局域网 .....	457
11.6 康柏ALPHA系列 .....	422	12.7.2 以太网 (CSMA/CD) 总线 .....	457
11.6.1 指令和寻址方式的格式 .....	422	12.7.3 令牌环 .....	458
11.6.2 ALPHA 21064处理器 .....	423	12.7.4 工作站网络 .....	458
11.6.3 ALPHA 21164处理器 .....	423	12.8 共享存储器和消息传递实例 .....	458
11.6.4 ALPHA 21264处理器 .....	423	12.8.1 共享存储器实例 .....	459
11.7 Intel IA-64系列 .....	424	12.8.2 消息传递实例 .....	461
11.7.1 指令包 .....	424	12.9 性能因素 .....	462



12.9.1 Amdahl定律 .....	463	附录B ARM指令集 .....	519
12.9.2 性能指标 .....	464	附录C Motorola 68000 指令集 .....	533
12.10 结束语 .....	464	附录D Intel IA-32指令集 .....	549
习题 .....	465	附录E 字符编码与数的转换 .....	565
参考文献 .....	467	索引 .....	569
附录A 逻辑电路 .....	469		

# 计算机的基本结构

### 本章目标

在本章中你将学习以下内容：

- 计算机的基本结构
- 机器指令及其执行
- 用于准备和执行程序的系统软件
- 计算机系统性能问题
- 计算机发展史

1

本书讲述的是计算机的组成结构。书中描述了数字计算机中用于存储和处理信息的各个部件的功能和设计，还介绍了与计算机相连的从外部设备接收信息和向外部指定设备传送计算结果的执行部件。本书的大部分内容是专门针对计算机硬件和计算机体系结构的。计算机硬件由电子电路、显示器、磁性和光存储介质、电动机械设备以及通信设施等构成，计算机体系结构包含具体的指令系统和用于执行这些指令的硬件设备。

本书还介绍了计算机系统中有关程序设计和软件构件方面的许多知识。要想对计算机系统有一个良好的认识，重要的是对各个计算机部件设计中的硬件和软件都要有所认识。

本章介绍了许多硬件和软件的概念，给出了一些通用技术，也对这门学科的基本内容作了概述。后面的章节会有更详细的讨论。

### 1.1 计算机的类型

首先对数字计算机（简称计算机）这一术语作一个定义。一般地，现代计算机是指一个可以接收数字输入信息，根据一连串内部存储指令对其进行处理，然后产生输出结果的高速电子计算机。其中的指令序列称为计算机程序，而内部存储称为计算机存储器。

各种类型的计算机在其大小、成本、计算能力和使用目的上有着很大的不同。在家庭、学校及办公室中最常用的计算机是个人计算机。它是台式机中最普通的一种形式。台式机中包括处理和存储单元、图像显示和音频输出单元，还有一个可以便于放在家庭书桌或办公桌上的键盘。存储介质包括硬盘、光盘和软盘。轻便的笔记本电脑是更精简的个人计算机，它将所有部件都装进一个如公文包大小的独立单元中。而那些具有高级图形输入/输出处理功能的工作站，虽然与台式机大小类似，但计算能力却远远超过个人计算机。工作站经常在工程应用中使用，尤其是用在交互式设计工作中。

除了工作站以外，还有许多大型的、功能强大的计算机系统，从低端的企业系统、服务器到高端的大型计算机。企业系统，或称为主机，通常用于大公司中的业务数据处理，因此需要比工作站更强的计算能力和更大的存储容量。服务器包含大容量的数据库存储部件，从而能够满足大量的数据访问需要。大多数情况下，服务器广泛地用于教育、商业以及个人用户群之中。请求和应答通常通过因特网的通信设施进行传送。实际上，因特网以及与其相关的服务器已经成为全球各类信息的主要来源。因特网通信设施包含复杂的高速光纤主干结构，它与广播电缆和电话线互连，从而构成与学校、商店、家庭的相连。

大型计算机用于需要进行大规模数字计算的工作中，如天气预报、飞机设计及仿真系统。在企业系统、服务器和巨型计算机中，各功能部件（包括多处理器）通常是由许多独立的、大型的部件构成。

## 1.2 功能部件

计算机包括五个功能相对独立的主要部分：输入设备、存储器、算术逻辑部件、输出设备、控制器，如图1-1所示。输入设备接收来自操作员、电动机械装置（如键盘），或是通过数字通信线路连接的其他计算机上的代码信息。所接收的信息或存在计算机的存储器中供以后使用，或立即传送到算术逻辑电路上执行所需要的操作。这个处理步骤是由内存中的一个程序决定的。最后，处理的结果通过输出设备送回到外部设备中。所有这些动作都是由控制器控制和协调的。图1-1并没有显示出这些功能部件之间的连接。有关这些可以通过多种途径实现的连接，将贯穿于全书中进行讨论。前面提到的算术逻辑电路与主控制电路结合构成了处理器，输入设备和输出设备通常整体称为输入输出设备（I/O）。

现在仔细观察一下计算机处理信息的过程。把信息划分成指令和数据将为讨论提供方便。指令，或称为机器指令，是具体的命令，它们：

- 控制计算机与I/O设备之间的信息传送。
- 具体指定要执行的算术和逻辑操作。

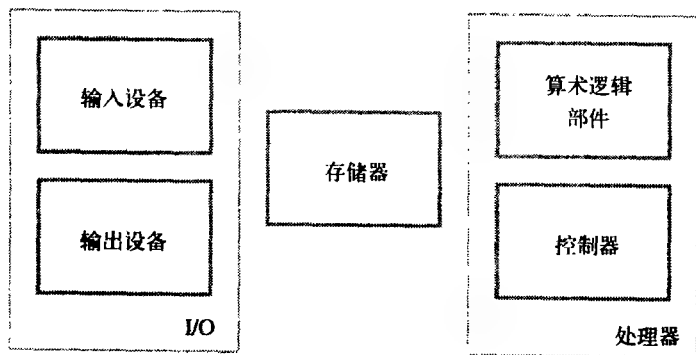


图1-1 计算机的基本功能部件

执行任务的指令序列称为程序。程序通常储存在内存中。处理器从内存中一个接一个地取出构成程序的指令，然后完成所需要的操作。除了可能有来自于操作员或与机器连接的I/O设备上的外部中断以外，计算机完全由存储程序（stored program）控制着。

数据是指作为指令操作数的数字和字符编码。它也经常指任意的数字信息。基于这个定义，

如果一个完整的程序（即一个指令序列）要被其他程序处理，那么这个程序可以被看作是数据。将高级语言的源程序编译成一连串的机器指令就是这样的一个例子，由机器语言构成的程序称为目标程序。源程序是编译程序的输入数据，编译程序将源程序翻译成机器语言程序。

计算机处理的信息必须按照适当的格式进行编码。当今的大多数硬件使用的是只有两个稳态的数字电路，即开和关（见附录A）。每个数字、字符或指令都被编码成叫做位的二进制数字串，它只有0和1两个可取的值。数字通常用按位二进制记数法表示，这些将在第2章和第6章中详细讨论。有时也使用二进制编码的十进制数（BCD），其中每一个十进制数都被编码成四位的二进制数。

字母数字字符也用二进制码表示。现已开发出了多种编码方案，最常用的两种是ASCII码（每个字符用七位二进制码表示）和EBCDIC码（扩展的二进制码的十进制数），其中每八位二进制数表示一个字符。有关二进制记数法和编码方案的更详细描述已在附录B中给出。

### 1.2.1 输入设备

计算机通过输入设备接收编码，即读取数据。大家最熟悉的输入设备就是键盘。当一个键被敲击时，相应的字母或数字就会自动被转换成相应的二进制码传送到内存或处理器中。

还有许多其他类型的输入设备，包括操作杆、跟踪球和鼠标。这些设备经常作为图形输入装置与显示器一起使用。话筒可以用来捕获声音输入，然后取样并转换为数字编码进行存储和处理。第10章将对输入设备及其操作作详细讨论。

### 1.2.2 存储器

存储器的功能是储存程序和数据。它分为主存储器和辅助存储器两种。

4

主存储器是以电子速度运行的高速内存。程序正在被执行时必须储存在内存中。内存包括大量的半导体存储单元，每一个能够存储一位二进制信息。这些单元很少被单独读取或编写，而是按固定大小的组进行处理，这个组称为字。内存规定包含有 $n$ 位的字的内容可以通过基本操作进行存储或检索。

为了在内存中能很容易找到任何一个字，每个字的单元都与一个不同的地址相关联。地址是用来识别逐个单元的数字。通过指明一个特定的字地址，然后给出一条开始进行存储或检索过程的控制命令，就能够对字进行访问了。

每个字的位数通常称作为计算机的字长。典型的字长范围是从16位到64位。内存的容量是衡量一台计算机规模的因素之一。小型的机器通常只有几千万字的容量，而中等或大型机一般有几千万或几亿字的容量。数据通常按单个字、多个字或字的部分单位进行处理。每访问一次内存，通常只有一个字的数据被读或被写。

在执行的过程中，程序必须保存在内存中。指令和数据既可以写入内存也可以在处理器的控制下读出。在内存中能够尽快地访问到任意字的位置是非常重要的。任何单元在指明了其地址后就能在很短而且是固定长度的时间内访问到的存储器，叫做随机访问存储器（RAM），访问一个字所需的时间叫做存储器访问时间。这个时间是固定的，与所访问的字的位置无关。当今RAM设备的访问时间通常在几纳秒（ns）至100纳秒之间。一台计算机的存储器可按照存储器层次结构，将半导体RAM设备根据不同的速度和大小分为3或4档。小而快的RAM设备称为高速缓存。它们紧挨着处理器并且通常由同一个集成电路芯片控制以达到高性能，最大的但最慢的设备是主存储器。稍后，在本章中我们将给出关于在存储器分层体系结构中信息访问的简要描

述。第5章将对计算机内存的操作及性能方面进行详细讨论。

虽然主存储器是必须的，但它的价格也是昂贵的。因此在需要存储大量数据和程序时，尤其是那些不经常访问的信息时就会使用附加并且比较便宜的辅助存储器。辅助存储器有很大的选择空间，包括磁盘、磁带和光盘（CD-ROM）等。这些设备也在第5章中做介绍。

### 1.2.3 运算器

5 大多数计算机的操作是由处理器的算术逻辑部件（ALU）执行的。考虑一个典型的例子：假设将内存中放置的两个数字求和。它们被送入处理器中，然后由ALU执行实际的加法操作。所求得的和可能储存在内存或保留在处理器中以便直接使用。

其他任意的算术或逻辑运算，例如乘、除，或比较数的大小，开始都是将操作数送至由ALU执行运算的处理器中。当操作数被送入处理器时，它们储存在高速的存储单元寄存器中。每个寄存器可以储存一个字的数据。在存储器层次结构中，寄存器的访问时间比最快的高速缓存的访问时间稍微快一点。

控制器和运算器比计算机系统连接的其他设备的运行速度要快许多倍。这就使得单独一个处理器能够控制许多的外部设备，如键盘、显示器，磁盘和光盘、传感器及机械控制器等。

### 1.2.4 输出设备

输出设备是输入设备的对应设备。它的功能是向外界输出处理结果。这类设备中最常见的是打印机。打印机使用机械冲击头、喷墨流或是激光打印机中的复印技术完成打印过程。现在可以生产出每分钟打印10 000行的打印机。这对机械设备来说是一个非常高的速度，但和处理器的电子速度相比仍然是很慢的。

一些设备，比如图形显示器，既有输出功能又有输入功能。这也是很多情况下对具有这种双重作用的设备使用单一名称“I/O设备”的原因。

### 1.2.5 控制器

6 存储器、运算器和输入输出设备对信息进行存储和处理，然后执行输入和输出操作。这些设备的操作必须按一定的方式互相协调。这就是控制器的工作。控制器是高效的中枢系统，它将控制信号传送到其他设备并检测它们的状态。

由输入和输出操作构成的I/O传输是被I/O程序的指令所控制的，I/O程序识别相关的设备和需要传输的信息。但是，控制传输的实际时序信号是由控制电路产生的。时序信号是决定何时发生规定动作的信号。处理器和存储器之间的数据传送也是由控制器通过时序信号控制的。于是有理由将控制器看作为一个非常明确而且物理上完全独立的设备，它和机器的其他部分有相互作用。但在实际中情况却恰恰相反。很多控制电路分布于整个机器中。大量的控制线（缆线）传递着所有部件中事件的时序和同步信号。

一台计算机的操作可以归纳如下：

- 计算机通过输入设备以程序和数据的形式接收信息，然后将其储存在存储器中。
- 在程序的控制下，存储器中的信息被取出，然后送入运算器中进行处理。
- 经过处理的信息由输出设备送出计算机。
- 机器内部的所有活动都由控制器控制。



### 1.3 基本操作概念

在1.2节中，我们介绍了计算机的活动是由指令控制的。为了执行一个给定的任务，要在存储器中储存一个包含一连串指令的相应程序。完成特定操作的指令从存储器中逐个取出，然后送入处理器，用作操作数的数据也储存在存储器中。一个典型的指令如下：

Add LOCA, R0

这条指令将存储器中位置为LOCA的操作数加到处理器中一个寄存器R0的操作数上，然后把所求的和放入寄存器R0中。原来LOCA位置中的内容被保护起来，而原来R0上的值被覆盖了。这条指令需要执行若干步。首先，指令从存储器中取出并被送入处理器中，然后，取出在LOCA处的操作数加到R0的操作数上。最后，所加的结果储存到寄存器R0中。

前面的加法指令将存储器存取操作与ALU操作结合在一起。在许多现代计算机中，由于性能的原因，这两种类型的操作被不同的指令执行，这个问题将在第8章中讨论。上述指令的效果可以由下面两条连续指令序列实现：

Load LOCA, R1

Add R1, R0

第一条指令将存储器中单元为LOCA的内容传送到处理器寄存器R1中，第二条指令将寄存器R1和R0中的内容相加然后将和送入R0。注意，这样同时破坏了寄存器R1和R0中的原始值，但存储器中单元为LOCA的原始内容被保存了下来。

存储器和处理器之间的传送从发送存储单元地址访问存储器并产生适当的控制信号开始，然后将数据送入或送出存储器。

图1-2显示了存储器和处理器是如何连接在一起的，也显示了一些还没有讨论过的处理器的重要操作细节。这些部件的互连模式并没有明确表示出来，因为到此为止我们只讨论了它们的功能特性。第7章将互相连接作为处理器设计的一部分进行了详细描述。

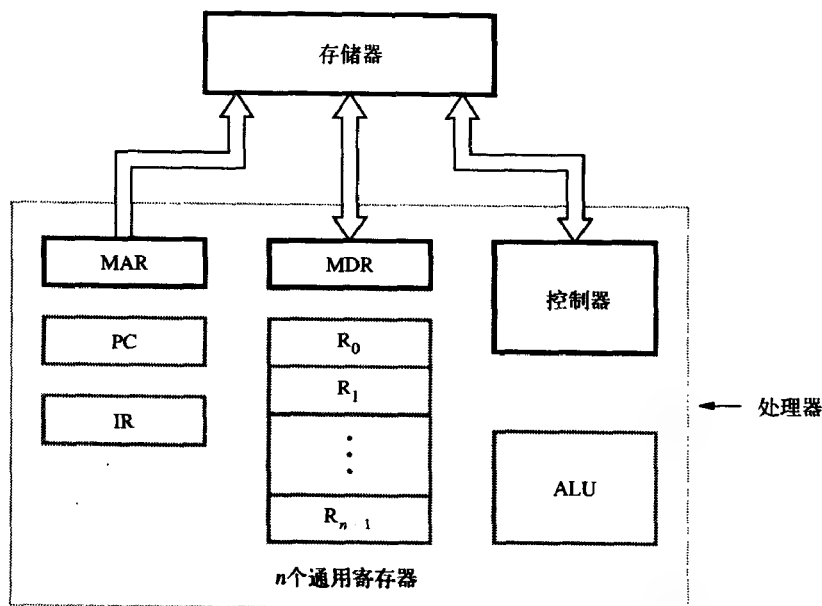


图1-2 处理器和存储器之间的连接

除ALU和控制电路外,处理器中包含许多用于不同目的的寄存器。指令寄存器(IR)保留当前正在执行的指令。它的输出结果由控制电路来获得,可产生能够控制执行指令中不同处理部件的时序信号。程序计数器(PC)是另一个有专门用途的寄存器。它跟踪程序执行的过程,其中包含下一条即将被读取和执行指令的内存地址。在一条指令执行的过程中,PC中的内容相应地被下一条将被执行指令的地址更新。习惯上说PC指向下一条将从内存中取出的指令。除了IR和PC,图1-2给出了 $n$ 个从 $R_0$ 到 $R_{n-1}$ 的通用寄存器。它们的作用将在第2章中解释。

还有两种寄存器与内存通信。这就是内存地址寄存器(MAR)和内存数据寄存器(MDR)。MAR保存着即将访问单元的地址。MDR保存着将写入该地址单元或从该地址单元中读出的数据。

现在来看一下典型的操作步骤。程序保存在内存中,而且通常是通过输入设备进行输入的。当PC指向程序开始的第一条指令时,程序开始执行。PC中的内容传送到MAR中,然后将一条读取控制信息送到存储器中。经历了访问内存所需的时间后,该地址字(这里指程序的第一条指令)从内存中读出并放进MDR中。接下来,MDR中的内容传送给IR。此时,指令已经准备好可以进行译码并执行了。

如果指令是一个有关ALU执行的操作,那么它就需要获取所需要的操作数。如果操作数保存在内存中(它也可以保存在处理器的一个通用寄存器中),就必须通过将它地址传送给MAR来获取,并且开始一个Read周期。当这个操作数已经从内存中读出并送至MDR时,它会从MDR传送到ALU中。当一个或是多个操作数按照这种方式取出后,ALU就可以执行所需要的操作了。如果这个操作结果将储存在内存中,那么该结果会送到MDR中。操作结果存储单元的地址将送往MAR中,并且开始一个Write周期。在当前指令执行过程中的某个点上,PC中的内容递增,以便使PC指向下一条要执行的指令。这样,一旦当前的指令执行完毕,新指令的读取就开始了。

除了在存储器和处理器之间传送数据外,计算机还从输入设备接收数据以及向输出设备输出数据,从而出现了一些具有处理I/O传送功能的机器指令。

当一些设备需要紧急服务时,程序的正常执行可能会被中断。例如,在一个计算机控制的工业流程中,监视器可能观察到了一个危险的情况。为了立即处理这一情况,必须中断当前程序的执行。为此,设备发出一个中断信号。所谓中断是来自I/O设备的需要处理器提供服务的请求。处理器提供请求所需要的中断服务程序(interrupt-service routine)。因为这样的转变可能会改变处理器内部的状态,所以必须在中断之前保存状态在存储单元中。通常,PC中的内容、通用寄存器和一些控制信息都储存在内存中。当中断服务程序完成时,处理器的状态就被恢复,从而使得被中断的程序可以继续执行。

图1-2中所示的处理器单元通常在一块超大规模集成电路(VLSI)芯片上实现,在同一块芯片上至少包含有存储器层次结构中的高速缓存部件。

## 1.4 总线结构

前面,我们已经介绍了计算机中各个部件的功能。为了构成一个可操作的系统,这些部件必须有组织地以某种方式连接起来。可以有多种方法实现这一点。在这里只考虑最简单而且最常用的方法。

为了达到一个合适的操作速度,计算机必须是有组织的,这样才能使所有单元在规定的时间内处理一个完整的数据字。当数据中的一个字在单元中传递时,它的所有位都在并行传递,

即这些位同时许多线路上传递，一位一条线。一组作为连接通道为多个设备服务的线路称为总线。除了传递数据的线路外，总线中还必须包括地址线和控制线。

各功能部件互连的最简单方法是使用单总线，如图1-3所示。所有的设备都和该总线相连。因为总线一次只能进行一种传递，所以只有两个设备可以在任何给定的时间内自由使用总线。总线的控制线是用来仲裁总线的多重使用需求的。单总线的主要优点是它的低功耗和连接外围设备的灵活性。包含多个总线的系统由于允许两个或更多的传递同时进行，因而在操作中具有更强的并行性。这样做获得了更好的性能，但却增加了功耗。

9

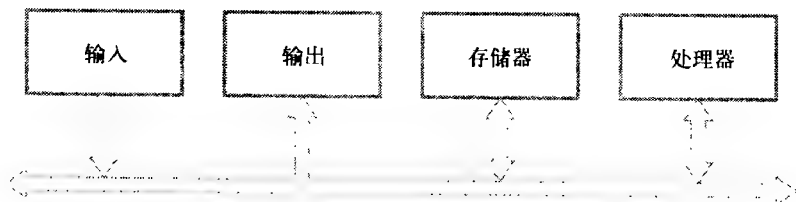


图1-3 单总线结构

连接在总线上的设备运行速度差别很大。一些电动机械设备，如键盘和打印机，速度相对来说比较慢。其他如光盘和磁盘，速度就相对很快。存储器和处理器以电子速度运行，它们是计算机中最快的部件。因为所有这些设备都要通过总线和其他的设备联系，所以需要有一个高效的传输机制，它不能被低速设备约束同时又能够调节处理器、内存和外部设备之间节奏上的差异。

一个常用的方法是设备带有缓冲寄存器，在传输过程中保存信息。为了举例说明这一技术，现在考虑一个字符编码从处理器到字符打印机的传输过程。处理器将字符通过总线传送到打印机的缓冲器中。因为缓冲器是一个电子寄存器，所以这个传送过程只需要相对短的时间。一旦缓冲器被装满，打印机就开始打印而不需要处理器的进一步干涉。此时不再需要总线和处理器，可以将它们释放出来去完成其他的活动。打印机继续打印在缓冲器中的字符，并且不接受新来的传输，直至这一打印过程结束。这样，缓冲寄存器就解决了处理器、内存和I/O设备间的节奏差异。这样做防止了在一系列数据传输中高速的处理器被锁定在低速的I/O设备上。这也使处理器能够快速地从—个设备转向另一个设备，在需要多个I/O设备的数据传输中交替进行处理活动。

## 1.5 软件

为了让用户能够输入并运行一个应用程序，计算机必须事先在其内存中存入一些系统软件。系统软件是一个程序的集合，其作用如下：

10

- 接收和解释用户的命令。
- 输入和编辑应用程序，然后将它们作为文件储存在辅助存储设备中。
- 管理辅助存储设备中的文件存储和检索。
- 运行由用户提供数据的标准应用程序，如字处理器、电子表格、游戏等。
- 控制I/O设备，接受输入信息并产生输出结果。
- 将用户制作的源程序格式转换成由机器指令构成的目标程序格式。
- 使用已存在的标准库程序，如数值计算包；链接和运行用户编写的应用程序。

系统软件就是这样对计算机系统的所有活动进行协调的。本节的目的介绍系统软件的

一些基本内容。

应用程序通常使用高级程序设计语言如C、C++、Java或Fortran编写，程序员在程序中指定数学的或文本处理的操作。这些操作使用一种格式进行描述，这种格式是独立于指定运行该程序的计算机的。使用高级语言的程序员不需要了解机器语言指令的细节。一个叫做编译器的系统软件程序可以将高级语言程序转换成恰当的机器语言程序，这些机器语言程序中包含指令，比如在1.3节中讨论过的Add和Load指令。

另一个所有程序员都要使用的重要系统程序是文本编辑器。它用来输入和编辑应用程序。这个程序与用户交互地执行命令，这些命令允许从键盘键入的源程序语句存放在文件中。文件是一系列储存在内存或辅助存储器中的简单字母、数字、字符或二进制数据。一个文件可以通过用户选择的名称进行访问。

本书中不讨论编译器、编辑器或文件系统的细节，下面来仔细地看一看被称为操作系统的主要系统软件构件。这是一个大型程序，实际上是一个程序的集合，它用来控制各种计算机设备在执行应用程序时的资源共享和相互作用。OS程序按要求执行为每个独立的应用程序分配计算机资源的工作。这些工作包括为程序和数据文件分配内存和磁盘空间、在内存和磁盘之间传递数据及处理I/O操作等。

为了理解操作系统的基本概念，现在假设系统只有一个处理器、一个磁盘和一台打印机。首先我们讨论运行一个应用程序的步骤。一旦解释了这些步骤，读者就会明白操作系统是如何处理多个应用程序同时执行的问题。假设这个应用程序已经从高级程序语言编译成机器语言并储存在磁盘中。第一步是将这个文件传送到内存中，当传输结束时，程序的执行就开始了。假设这个程序的一部分工作是从磁盘中读取一个数据文件并送入内存，对数据进行一些运算，然后打印出结果。当需要数据文件时，程序就会请求操作系统将磁盘中的数据文件传输到内存中。OS完成这项工作并将执行控制返还给应用程序，然后应用程序继续进行所需要的运算。当运算完成并准备打印结果时，应用程序再一次向操作系统发出请求，于是有一个OS程序使打印机打印出结果来。

我们已经看到执行控制权是如何在应用程序和OS程序之间来回传递的。使用一个时间线路图可以方便地说明处理器执行时间的共享问题，如图1-4所示。在  $t_0$  至  $t_1$  段，OS程序开始将应

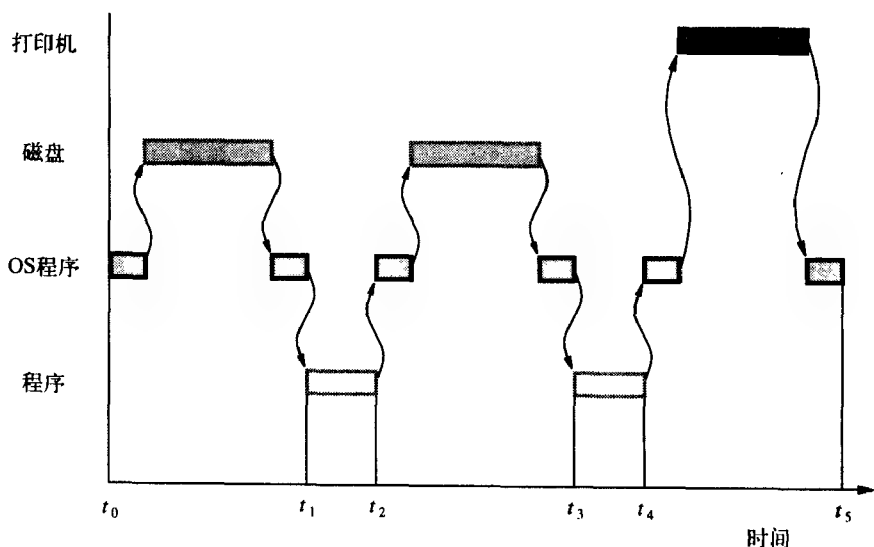


图1-4 用户程序和OS程序共享处理器

用程序从磁盘装载到内存，等待装载传输结束后将执行控制权传送给应用程序。类似的操作也发生在  $t_2$  至  $t_3$  段、 $t_4$  至  $t_5$  段，即当操作系统从磁盘中传递数据文件时和打印结果时。在  $t_5$  时刻，操作系统可能已经在装载和执行另一个应用程序了。

现在介绍当多个应用程序需要处理时能高效使用计算机资源的方法。我们注意到磁盘和处理器在  $t_4$  至  $t_5$  段的大部分时间内是闲置的，操作系统可以在打印机正在工作的时刻就从磁盘中将下一个要执行的程序装入到内存中。同样，从  $t_0$  至  $t_1$  段，操作系统可以在当前程序正在从磁盘装入时安排打印前一程序的结果。这样，操作系统协调多个应用程序的并行执行，这样可以最大限度地利用计算机资源。这种并行执行的方式称为多道程序或多任务。

12

## 1.6 性能

衡量一台计算机性能好坏的最主要因素是看它执行程序的速度有多快。一台计算机执行程序的速度受到硬件及机器语言指令设计的影响。由于程序常常使用高级程序设计语言编写，所以性能也会受到将程序转换成机器语言的编译器的影响。为了达到最佳性能，需要使编译器、机器指令设备及硬件的设计协调一致。本书不描述编译器设计的细节，只关注指令设备和硬件的设计。

在1.5节中，我们描述了操作系统是如何进行重叠处理、磁盘传输以及最大限度使用可用资源处理多个程序打印等问题。在图1-4中给出的执行程序所需要的总时间为  $t_5 - t_0$ 。这一段实际消耗时间是对整个计算机系统的性能测量。它受到处理器、磁盘和打印机速度的影响。描述处理器的性能，应该只需要考虑处理器正在工作的时间，即在图1-4中标注有程序及OS程序的时间段。这些时间段的总和称为程序执行所需的处理器时间。接下来了解一些影响处理时间的主要参数，并指出讨论相关问题的章节，希望读者在学习后面内容时能时刻记起这些关于性能的描述。

就像执行一个程序实际消耗的时间依赖于计算机系统的所有部件一样，处理器时间依赖于执行每条机器指令的硬件。这些硬件包括处理器和存储器，它们通常由一条总线连接起来，如图1-3所示。图中相关的部分在图1-5中再一次表示出来，包含作为处理器单元中一部分的高速缓存（cache）。现在让我们来看一下程序指令和数据在内存与处理器之间的流动过程。在开始执行时，所有程序指令和所需数据都存储在主存中。随着执行过程的推进，指令经过总线被逐条提取到处理器中，并且在高速缓存中存放了一个副本。当指令的执行需要主存储器中的数据时，数据被取出并在高速缓存中存放一个副本。以后，如果需要再一次使用相同的指令或数据时就可以直接从高速缓存中读出。

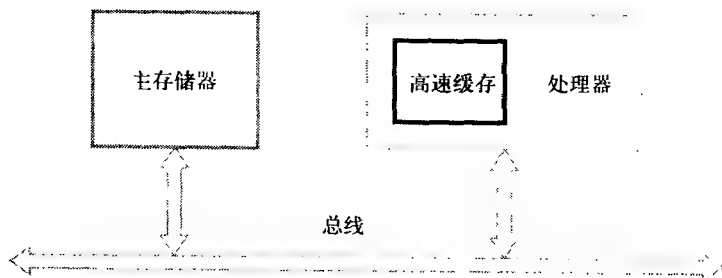


图1-5 处理器的高速缓存

13

处理器和相对小的高速缓存可以集成在一个单独的集成电路芯片上。这类芯片执行指令处理基本步骤的内部速度非常快，比从主存储器中提取指令和数据要快得多。主存和处理器之间



的指令及数据的移动时间越短, 程序就执行得越快, 使用高速缓存可达到这一目标。例如, 假设许多指令在一段很短的时间内重复执行, 比如程序中的循环语句。如果这些指令可在高速缓存中获得, 它们可以在重复使用时被很快地取出来。这种方式同样适用于数据的重复使用。主存储器 and 高速缓存的设计、操作和执行问题将在第5章中讨论。

### 1.6.1 处理器时钟

处理器电路由称为时钟的时序信号控制。时钟定义了规则的时间间隔, 称为时钟周期。为了执行一条机器指令, 处理器将要被执行的动作分解为一系列基本步骤, 这样使每个步骤都可以在一个时钟周期内完成。时钟周期的长度 $P$ 是一个影响处理器性能的重要参数。它的倒数是时钟频率 $R = 1/P$ , 使用 $R$ 可以度量每秒的周期数。现今的个人计算机和工作站中使用的处理器的时钟频率范围从每秒几亿次到十亿次以上。在标准的电子工程术语中, 将“每秒的周期数”称为赫兹 (Hz)。“百万”用前缀M表示, “十亿”用前缀G表示。因此, 每秒5亿个周期通常表示为500 MHz, 每秒12.5亿个周期表示为1.25 GHz。相应的时钟周期分别为2纳秒 (ns) 和0.8纳秒。

### 1.6.2 基本性能公式

下面来关注处理器总消耗时间的构成。令 $T$ 为执行使用高级语言编写的程序所需要的处理器时间。编译器生成一个与源程序相符的机器语言目标程序。假设该程序的完整执行需要执行 $N$ 个机器语言指令。数字 $N$ 为实际指令执行数, 它不必与目标程序中的机器指令数相等。有些指令可能不止一次被执行, 就像在循环程序中指令的情况。其他的指令可能根本不被执行, 这取决于使用的输入数据。假设执行一条机器指令需要的基本步骤平均数为 $S$ , 每个步骤都在一个时钟周期内完成。如果时钟频率为每秒 $R$ 个周期, 则程序执行的时间为:

$$T = \frac{N \times S}{R} \quad (1-1)$$

这个公式常被称为基本性能公式。

对于用户来说, 应用程序中的性能参数 $T$ 比参数 $N$ 、 $S$ 或 $R$ 任一个都重要得多。为了达到较高的性能, 计算机设计人员必须找到减少 $T$ 值的方法, 这意味着减少 $N$ 和 $S$ , 增加 $R$ 。如果源程序被编译成较少量的机器指令,  $N$ 的值就会减少。如果指令包含更少需要执行的基本步骤或者指令的执行是重叠的,  $S$ 的值就会减少。使用高频时钟会增加 $R$ 的值, 这意味着完成一个基本执行步骤的时间减少了。

必须强调 $N$ 、 $S$ 和 $R$ 不是独立的参数; 改变一个就有可能影响到其他两个值。在处理器设计中引入一种新特征, 只有在总的结果是减少了 $T$ 值时才能够实现性能改善。一个具有900MHz时钟的处理器未必比一个700MHz的处理器性能好, 原因是它们可能有着不同的 $S$ 值。

### 1.6.3 流水线和超标量操作

在上面的讨论中, 我们假设指令是一个接一个被执行的。因此,  $S$ 是执行一条指令总的基本步骤数, 或时钟周期数。使用一种称为流水线的技术, 可以通过叠加连续指令执行的方法真正地改善性能。考虑下面的指令

Add R1, R2, R3

该指令将寄存器R1和R2的内容相加, 并将结果放到R3中。R1和R2中的内容首先被传送到ALU

的输入中,在加法运算执行之后,其和被送到R3中。处理器可以在执行加法操作时从内存中读取下一条指令。然后,如果该指令仍然使用ALU,它的操作数可以在上一条求和指令的结果送到R3的同时传送至ALU的输入中。理想情况下,如果所有的指令都最大限度地被叠加,执行过程以每个时钟周期完成一条指令的速度进行。个别的指令仍然需要几个时钟周期去完成。但是为了计算 $T$ 值, $S$ 的有效值为1。

流水线技术将在第8章中做详细讨论。正如我们将要看到的,由于各种原因不能够在实际中达到理想值 $S=1$ 。但是,流水线明显地提高了指令执行的速度,并使 $S$ 的有效值接近于1。

如果在处理器中实现了多个指令流水线就能达到更高度的并发性。这意味着使用多功能部件创造了可以使不同指令并行执行的平行路线。在这样的安排下,有可能在每个时钟周期中开始若干条指令的执行。这种运算方式称为超标量执行。如果在程序执行中能长时间保持这种运算方式, $S$ 的有效值可减小到1以下。当然并行执行必须保证程序逻辑的正确性,也就是说,产生的结果必须与串行执行程序指令的结果相同。现今许多高性能处理器就是基于这种操作方式设计的。

15

#### 1.6.4 时钟频率

提高时钟频率 $R$ 有两种可能的方法。第一,改进集成电路(IC)技术,减少完成一个基本步骤所需要的时间,从而使逻辑电路速度加快。这样就使时钟周期 $P$ 缩短,时钟频率 $R$ 提高。第二,减少在基本步骤中的处理量也可以缩短时钟周期 $P$ 。但是,如果一条指令要求执行的动作必须保持不变,所需的基本步骤数可能会增加。

$R$ 值的提高完全取决于IC技术的改进,它影响了处理器操作的各个方面,而不仅仅是对访问主存时间的影响。当存在高速缓存时,访问主存所占的百分比很小。因此,多数指令执行从使用可实现的快速技术中获得了预期的效果。当 $R$ 增加时由于 $S$ 和 $N$ 不受影响, $T$ 值也将随之减小。改变指令划分基本步骤的方法对性能的影响更加难以估计。这个问题将在第8章中讨论。

#### 1.6.5 指令集: CISC和RISC

简单的指令需要执行少量的基本步骤。复杂的指令则需要大量的步骤。对于一个只有简单指令的处理器来说,一项给定的程序设计任务可能需要执行大量的指令。这可能导致一个大的 $N$ 值和一个小的 $S$ 值。另一方面,如果单独的指令执行了较多的复杂操作,需要的指令就少,这又会导致一个小的 $N$ 值和一个大的 $S$ 值。哪种选择更优越并不明显。

比较两种选择的关键因素是流水线的使用。前面已经指出流水线处理器中的 $S$ 有效值接近于1,即使每个指令的基本步骤值可能相当大。这就预示着与流水线结合的复杂指令可以达到最佳性能。但是,使用简单指令集在处理器中实现高效流水线更加容易。在流水线执行中选择合适的指令集是一个重要的因素,而且常常是决定性的。

16

指令集的设计和可用的选择将在第2章中讨论。关于简单指令处理器和较复杂指令处理器各自的优点我们已经了解了很多<sup>[1]</sup>。前者被称作为精简指令集计算机(RISC),后者被称为复杂指令集计算机(CISC)。在第3章和第11章中给出了RISC和CISC的例子,并且讨论了它们的优点。虽然为了与当前的描述相一致我们使用了术语RISC和CISC,但提醒读者不要认为它们是清晰的计算机分类。现有的处理器设计是多方面权衡的结果。术语RISC和CISC涉及到的设计原则和技术会在本书中多次进行讨论。

### 1.6.6 编译器

编译器将高级语言设计的程序转换为一系列机器指令。为了减少 $N$ 值，需要有一个合适的机器指令集和能很好使用它的编译器。优化编译器利用各种目标处理器的特性来减少 $N \times S$ 的值，这一乘积表示执行一个程序所需的时钟周期总数。在第8章中将会看到周期数不仅仅取决于指令的选择，还取决于它们在程序中出现的顺序。编译器会重新安排程序指令以获得更高性能。当然，这些变化不会影响到计算结果。

表面上看，编译器是一个与处理器独立使用甚至可能从不同的销售商手中买到的实体。但是，一个高质量的编译器一定与处理器体系结构紧密相关。编译器和处理器常常同时设计，并通过设计者的相互协调来达到最佳结果。最终的目的是为了在执行程序设计任务时，所需的时钟周期总数减少。

### 1.6.7 性能测量

对一台计算机性能的评估是很重要的。计算机设计者要使用性能评估来对新功能的有效性进行评价。制造商在营销过程中要使用性能指标。购买者要使用这些数据在众多的计算机型号中进行选择。

这些讨论使人想到描述计算机性能的惟一合适参数是对程序的执行时间 $T$ 。尽管公式(1-1)中的概念很简单，计算 $T$ 的值却不那么容易。此外，像时钟速度和各种体系结构特性这样的参数并不能对预期的性能做出可靠指示。

[17]

基于这些原因，计算机协会提出使用基准程序来测量计算机性能的思想。为了能够进行比较，必须使用标准化的程序。这个性能测量的是计算机执行一个规定的基准程序所需要的时间。最初，设计者试图创造一些假想的程序作为标准的基准。但是假想程序不能恰当地预测实际应用程序运行时所能达到的性能。

现今公认的方法是使用一个一致的实际应用程序来评估性能。一个叫做系统性能评估协会(SPEC)的非营利组织选择并公布不同应用领域中具有代表性的应用程序以及许多商业应用计算机的测试结果。1989年为通用型计算机选择了一套基准程序。1995年和2000年先后对它进行了少量的修改，并做了再版。

程序的选择范围从游戏比赛、编译程序、数据库应用到天体物理学和量子化学中的数字集约程序。每种情况下，程序都在被测试的计算机上编译并且得出在实际计算机上的运行结果(不允许仿真)。在选择的一台作为参照的计算机上同样的程序也要被进行编译和运行。对于SPEC95，参照的是SUN SPARCstation 10/40。对于SPEC2000，参照的是使用300MHz UltraSPARCIIi处理器的UltraSPARC 10工作站。SPEC的等级计算如下：

$$\text{SPEC的等级} = \frac{\text{在参照计算机上的运行时间}}{\text{在被测试计算机上的运行时间}}$$

如果SPEC的等级是50，表示在该特定基准下，被测计算机运行速度是UltraSPARC 10的50倍。在SPEC组内对所有程序测试是重复进行的，并且计算的结果是几何平均的。令 $\text{SPEC}_i$ 为组中第 $i$ 个程序的等级。则计算机总的SPEC等级为：

$$\text{SPEC的等级} = \left( \prod_{i=1}^n \text{SPEC}_i \right)^{\frac{1}{n}}$$

这里 $n$ 为组中的程序数。

因为实际的执行时间被测量出来, SPEC等级是影响性能的全部因素的组合效果的测量值, 包括编译程序、操作系统、处理器和被测计算机的存储器。关于SPEC基准程序和测试处理结果的细节可在SPEC网页<sup>[2]</sup>中找到。

## 1.7 多处理器和多计算机

至此, 我们所介绍的计算机是只有一个处理器的计算机。大型计算机系统中可能包含有许多处理器单元, 这种系统称为多处理器系统。这些系统中的每个处理器并行执行许多不同的应用任务或一个独立的大型任务中的子任务。在这样的系统中所有处理器通常有权利访问这种系统中的所有内存, 并且经常使用共享存储器多处理器系统来清楚地表明这一点。系统的这些高性能使得复杂性和成本大大增加。除了多处理器和存储器单元之外, 因为需要更多复杂的互连网络也使成本不断增加。

[18]

和多处理器系统相比, 使用一个完整计算机互连组来达到总的高计算能力也是可行的。这些计算机通常只访问它们自己的存储器, 当任务的执行中需要数据传递时, 它们用通信网交换信息的方式来实现。这一特点将它们与共享存储器的多处理器区别开来, 因此将它们命名为信息传递多计算机系统。

共享存储器多处理器和信息传递多计算机系统, 以及这类系统中的互连网络将在第12章中描述。

## 1.8 发展历程

我们现在所知道的计算机已经有超过60年的历史。在计算机被发明之前有一段漫长的机械计算设备发展的过程。这段历史有很多的原始描述。例如, Hayes<sup>[3]</sup>非常好地描述了计算机的历史, 包括日期、发明者、设计者、研究机构和制造商。这里, 我们只简要地描述一下计算机发展的历程。

在20世纪中叶之前的300年里, 一系列由齿轮、杠杆和滑轮构成的越来越复杂的机械装置, 被用来做基本的加、减、乘、除操作。穿孔卡片上的孔被机器感知后自动控制一系列的计算, 这是当时提供的主要编程能力。这些设备可以计算完整的对数表和使用多项式近似的三角函数表, 输出结果被穿孔在卡片上或打印在纸上。在第二次世界大战期间, 电动机械传动设备, 比如像使用在早期电话转换系统中的设备, 提供了计算机构造中执行逻辑功能的方法。同时, 第一台用于无线电和军事雷达设备中的基于真空管技术的电子计算机在宾夕法尼亚大学设计制造完成。真空管用来执行逻辑运算和存储数据。这项技术开创了电子数字计算机的新纪元。

计算机的处理器、存储器和I/O设备的制造技术的发展分为四代: 第一代是1945~1955年; 第二代是1955~1965年; 第三代是1965~1975年; 第四代是从1975年至今。

### 1.8.1 第一代计算机

程序存储的核心概念是由John von Neumann提出的。程序和程序中使用的数据被放在同一个存储器中, 就像今天这样。汇编语言用来准备程序并将它转换成可执行的机器语言。

使用真空管技术实现的逻辑功能, 基本算术运算只需要执行几毫秒。这比早期的机械和基于传动的电动机械技术的速度要快100~1000倍。最初使用的是镀银延迟线存储器, 而I/O功能

[19]

由类似打字机的设备完成，还开发出了磁心存储器和磁带存储设备。

### 1.8.2 第二代计算机

在20世纪40年代后期AT&T贝尔实验室发明了晶体管，并迅速用它取代了真空管。这一基础技术的转变标志着第二代计算机的开始。在第二代计算机中广泛使用了磁心存储器和磁鼓存储设备，开发出了像Fortran这样的高级语言，使得应用程序的制作更为容易。称为编译程序的系统程序也被开发出来，它将高级语言程序翻译成相应的汇编语言程序，这些汇编语言然后再被翻译成可执行的机器语言形式。在这一时期还开发出了独立的I/O处理器，它可以与中央处理器并行操作所执行的程序，从而提高了总的执行性能。在这一时期中，IBM公司成为了主要的计算机制造商。

### 1.8.3 第三代计算机

在一个独立的硅芯片上建造许多晶体管的能力即是集成电路技术，它可以建造低成本的高速处理器和存储单元。集成电路存储器开始取代磁心存储器。这一技术的发展标志着第三代计算机的开始。在这个时期还发展了其他技术，诸如采用的微编程、并行性和流水线技术。操作系统软件可以使若干个用户程序有效地分享计算机系统。还开发了高速缓存和虚拟存储技术。高速缓存使得主存储器看起来比实际要快，而虚拟存储使它看起来比实际的存储器要大。IBM公司的系统360主机和Digital Equipment（数字设备）公司的PDP系列小型机主宰了第三代计算机的商用产品市场。

### 1.8.4 第四代计算机

在20世纪70年代初，集成电路制造技术已经达到小型计算机中整个的处理器和大部分主存储器可以集成在单个芯片上的程度。数以万计的晶体管可以放置在一块芯片上，超大规模集成电路（VLSI）这一名词描述了这项技术。VLSI技术使得一个完整的处理器可以在一块芯片上制造，这就是微处理器。一些公司如Intel、National Semiconductor、Motorola、Texas Instruments和Advanced Micro Devices是这一技术的推动者。

诸如并行、流水线、高速缓存和虚拟存储器等组成结构的概念用在了当今第四代成熟的高性能计算系统的生产中。由局域网、广域网或因特网互联的笔记本电脑、台式个人电脑和工作站已经成为计算的主体方式。在主机上的集中式计算现在主要用于大型公司的商业应用中。

### 1.8.5 后第四代计算机

后第四代计算机有时用来描述一些具有支配机构或应用驱动能力的计算机系统。近年来，在描述这些逐步形成的系统时，倾向于使用这些特征而不是使用时代序号。人工智能型计算机、大型并行机、广域分布式系统是当今趋势的范例。或许最重要的是，计算机产业的发展依赖于功能日益强大并且大众支付得起的台式机，以及广泛使用的因特网上的大量信息资源。

### 1.8.6 性能的发展

从机械和电动机械设备向最初基于真空管的电子设备的转变带来了100~1000倍速度上的提高，从秒级提升到毫秒级。晶体管取代真空管的更替带来了另一个1000倍的速度提高，基本操

作可以在微秒内执行。集成电路制造中日益提高的密度产生了当今在毫微秒或更短的时间内执行基本操作的微处理器芯片，又使速度有了1000倍的提高。除了技术上的发展，在计算机体系结构上还产生了许多新方法，比如对计算机性能有显著影响的高速缓存和流水线技术的使用等。

## 1.9 结束语

这一章介绍了计算机结构和操作方面的许多内容。对许多与主题有关的术语都作了介绍，也概述了一些重要的设计概念。后面的章节将对这些术语和概念做出完整的解释，还会对本章各部分的内容做适当的分析。

21

## 习题

1.1 按照图1-2中所示的部件之间的数据项传递和一些简单的控制命令列出下面机器指令所需步骤

Add LOCA, R0

假设指令本身存储在内存中的INSTR单元中，并且该地址开始在寄存器PC中。前两步可能为

- 将寄存器PC中的内容传送到寄存器MAR中。
- 给内存下达一条读的命令，然后等待它将请求送到寄存器MDR中。

注意包括需要从INSTR到INSTR+1的PC内容的更新，以便可以取出下一条指令。

1.2 对在1.6.3节中讨论的机器指令

Add R1, R2, R3

重复习题1.1。

1.3 (a) 对任务“将存储单元A和B中的内容相加，并将结果放到单元C中”给出一个机器指令的短序列。指令

Load LOC, R<sub>i</sub>

及

Store R<sub>i</sub>, LOC

是用于在内存和通用寄存器R<sub>i</sub>之间传递数据的惟一可用指令。Add指令在1.3节和1.6.3节中进行了描述。不要破坏单元A和B中的内容。

(b) 假设Move和Add指令可用格式有：

Move/Add Location1, Location2

这些指令传送或添加一个操作数副本从Location1到Location2，覆盖第2个位置中原有操作数内容。Location<sub>i</sub>既可以在内存中也可以在处理器寄存器中。是否可以使用较少的指令完成(a)部分中的任务？如果可以，请给出序列。

1.4 (a) 在1.5节中讨论了一个程序集合的输入输出步骤，如图1-4所示，可以进行重叠以减少总的执行时间。令每6个OS程序执行间隔为1个时间单位，其中每个磁盘操作需要3个单位，打印需要3个单位，每个程序执行间隔需要2个时间单位。计算对于一个长程序序列最佳重叠时间与不重叠时间的比。忽略开始和结束的瞬间时间。

22

(b) 1.5节说明程序计算可以分别与输入或输出操作重叠或是同时与两者重叠。忽略OS程序需要的相对较短的时间，对于一个程序集合整个的执行，最佳重叠时间与不重叠时间的比



是多少？每个程序在输入、计算和输出活动上如何平衡？

- 1.5 (a) 在1.6.2节提到的程序执行时间 $T$ 用于检测某些高级语言程序。程序可以在一台 RISC或 CISC计算机上运行。这两者都使用流水线指令执行，但是RISC机器上的流水线比CISC机器上的效率更高。具体地说， $T$ 表达式中 $S$ 的有效值在RISC机器上为1.2，但在CISC机器上只有1.5。两台机器有着相同的时钟频率 $R$ 。如果在CISC机器上与RISC机器上的执行时间一样长，那么 $N$ 的最大允许值（CISC机器上的指令执行数）与RISC机器上的 $N$ 值百分比是多少？
- (b) 当时钟频率 $R$ 在 RISC机器上比CISC机器上快15%时，重复(a)部分中的问题。
- 1.6 (a) 在1.6节中讨论过，如图1-5所示的一个处理器高速缓存。假设一个程序的执行时间正好与指令访问时间成比例，并且在高速缓存中访问一条指令比在主存中访问一条指令快20倍。假设指令在高速缓存中的概率为0.96，还假设如果指令在高速缓存中没有找到，必须首先将它从主存中读取出来送到高速缓存中，然后从高速缓存中取出再被执行。计算没有高速缓存与有高速缓存情况下程序执行时间的比值。这个比值通常定义为由于存在高速缓存而获得的加速因子。
- (b) 如果高速缓存的大小加倍，同时假设在其中找不到所需指令的概率减半，对于双倍大小的高速缓存重复(a)部分中的问题。

## 参考文献

1. D.A. Patterson and J.L. Hennessy, *Computer Organization and Design — The Hardware/Software Interface*, 2nd ed., Morgan Kaufmann, San Mateo, Calif., 1998.
2. System Performance Evaluation Corporation web page: [www.spec.org](http://www.spec.org).
3. J.P. Hayes, *Computer Architecture and Organization*, 3rd ed., McGraw-Hill, New York, 1998.

# 机器指令和程序

### 本章目标

在本章中你将学习以下内容:

- 机器指令和程序的执行, 包括程序转移、子程序调用以及返回操作
- 数的表示以及在二进制补码系统中的加法/减法
- 访问寄存器和存储器的操作数寻址方式
- 用于表示机器指令、数据和程序的汇编语言
- 程序控制的输入/输出操作
- 基于堆栈、队列、列表、链表和数组等数据结构的操作

25

这一章我们从机器指令集的角度来考虑程序在计算机中的执行方法。第1章已经介绍了存储在内存中的程序指令和数据操作数的一般概念, 本章将学习指令序列从存储器传递到处理器, 并完成给定任务的方法。寻址方式通常用于访问在内存单元和所使用的处理器寄存器中的操作数。

这里强调的是基本概念。我们用一种通用的方式描述机器指令和操作数的寻址方式, 这种方式是商用处理器的典型方式。本章将介绍足够的指令和寻址方式, 以便于能够给出一个针对简单任务的完整且真实的程序。这些通用程序用汇编语言进行了说明。在汇编语言中, 机器指令和操作数寻址信息用符号名来表示。完整的指令集通常称为处理器的指令集体系结构 (ISA, instruction set architecture), 其中除了对指令做具体说明外, ISA中还详细说明了用于访问数据操作数和该指令用到的处理器寄存器的寻址方式。在本章中我们对基本概念进行讨论, 不需要定义完整的指令集, 因此所采用的方式是给出足够的例子去说明所需要的性能。

第3章介绍了由ARM、Motorola和Intel公司生产的三种商用处理器的ISA, 本章中的通用程序在第3章中按照这三种指令集再次分别给出, 为读者提供来自于实际机器的例子。

大多数的程序是用高级语言编写的, 比如用C、C++、Java或Fortran。在本书中使用汇编语言编程主要是为了描述计算机是如何操作的。为了在处理器上执行高级语言程序, 这个程序必须首先被解释成该处理器上的汇编语言。汇编语言是机器语言的一种易读形式。高级语言与机器语言之间的特征关系是计算机设计中需要考虑的一个关键问题, 我们也将多次讨论。

所有的计算机都可以处理数字, 它们具有执行数据运算对象的基本算术运算指令。同样, 在一个程序的机器指令执行过程中, 需要运行算术操作去生成一些数字, 这些数字表示在存储器中被访问的操作数单元的地址。要了解这些任务是如何实现的, 必须首先知道数字在计算机

中是如何表示的, 以及这些数字在做加法和减法运算时是如何被操作的。因此, 在本章的第1节我们将介绍这些主题。有关实现计算机算术运算的逻辑电路将在第6章详细讨论。

除了数字数据外, 为了处理文本信息计算机还要处理字符和字符串。本章的第1节描述了字符在计算机中是如何表示的。

26

## 2.1 数、算术运算以及字符

计算机建立在用二值电信号(参见附录A)表示信息的逻辑电路的基础上。我们把这两个值说成是“0”和“1”, 并且定义使用这种信号表示的信息量是一个信息位, 这里的一位表示一个二进制数。在计算机系统中表示数的最基本方法是使用一串位, 即一个二进制数。文本字符也可以用一串位表示, 叫做字符码。

我们首先介绍二进制数的表示和对这些数的算术运算, 然后描述字符的表示方式。

### 2.1.1 数的表示

考虑一个 $n$ 位的向量

$$B = b_{n-1} \cdots b_1 b_0$$

对于 $0 \leq i \leq n-1$ ,  $b_i = 0$ 或 $1$ 。这个向量在 $0$ 到 $2^n - 1$ 的范围内可以用无符号整数 $V$ 表示, 这里

$$V(B) = b_{n-1} \times 2^{n-1} + \cdots + b_1 \times 2^1 + b_0 \times 2^0$$

显然需要表示正数和负数, 有三种系统可以用来表示这些数:

- 原码
- 反码
- 补码

在这三种编码体系中, 最左边一位为“0”时表示正数, 为“1”时表示负数。图2-1用4位数举例说明了在这三个体系中数字的表示方法。在所有的编码体系中正数的表示法相同, 而负数有着不同的表示方法。在原码系统中, 负数值是将向量 $B$ 相应的正数值的最高有效位(在图2-1中的 $b_3$ )由0变为1来表示的。例如: +5表示为0101, -5表示为1101。在反码表示中, 负数是由相应的正数中的每一位的值求反而获得的。因此, 对于-3来说, 对向量0011的各位求反就得到了1100。显然, 运用同样的按位求反操作也可以把一个负数转换成相应的正数。这两种转换方法称为一个给定数的反码形式。对于一个给定数做反码形式的操作相当于从 $2^n - 1$ 中减去这个数, 在图2-1中的4位数情况下也就是从1111中减去该数。最后在补码系统中, 一个数的补码是从 $2^n$ 中减去这个数而得到的。

27

因此, 一个数的补码可以用这个数的反码加1而获得。

值得注意的是在原码和反码系统中对于“+0”和“-0”有着截然不同的表示, 而在补码系统中对“0”只有一种表示。对4位数来说, 在补码系统中对数值-8有表示, 而在其他系统中却没有。原码系统看似最接近自然数, 因为在笔算中我们就是用原码处理十进制数的。反码系统容易与这个系统关联起来, 而补码系统看起来就不自然了。但是, 我们将会在第2.1.3节中说明补码系统为进位加法和减法运算提供了最有效的方法。它是在计算机中最常用的方法之一。

$B$	值的表示		
$b_3b_2b_1b_0$	原码	反码	补码
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-7	-8
1 0 0 1	-1	-6	-7
1 0 1 0	-2	-5	-6
1 0 1 1	-3	-4	-5
1 1 0 0	-4	-3	-4
1 1 0 1	-5	-2	-3
1 1 1 0	-6	-1	-2
1 1 1 1	-7	-0	-1

图2-1 二进制有符号整数的表示

2.1.2 正数的加法

考虑两个1位数相加，这个结果在图2-2中给出。注意1加1的和需要一个2位向量10来表示数值2。我们说这个和是0，进位是1。为了完成多位数相加，使用一种类似于十进制数笔算中使用的方法。从这个位向量的最低位（右边）开始进行两位相加，并将进位传递到它的高位（左边）上去。

28

0	1	0	1
+ 0	+ 0	+ 1	+ 1
0	1	1	10
			↑
			进位

图2-2 1位数的加法

2.1.3 有符号数的加法和减法

我们介绍了表示正数和负数（或简称为有符号数）的三种系统。这些系统的不同之处仅在于对负数的表示方式。从执行算术运算容易性的观点来看它们的相应优势可以归纳如下：自然数表示法在表示上是最简单的，但是它对于加法和减法的运算也是最不便的。反码表示法稍微好一些，补码系统对于执行加法和减法运算是最有效的。

为了理解补码的算术运算，考虑以 $N$ 为模的加法（写成 $\text{mod } N$ ）。对于描述正整数 $\text{mod } N$ 加法的有效图示方式是使用一个具有 $N$ 个值的圆，0到 $N-1$ 是圆周上的标记，如图2-3a所示。考虑当 $N = 16$ 的情况， $(7+4) \text{ mod } 16$ 运算产生的值是11。为了使用图形执行这个操作，在圆上找到7的位置然后按顺时针方向移动4个单位就到达了答案11。类似地， $(9 + 14) \text{ mod } 16 = 7$ ；在圆上找到9然

后按顺时针方向移动14个单位就到达答案7。这种计算 $(a+b) \bmod 16$ 的图形技术，对于任何正数 $a$ 和 $b$ 都是有效的。即为了完成加法，定位 $a$ ，然后按顺时针方向移动 $b$ 个单位就可以得到 $(a+b) \bmod 16$ 的结果。

现在考虑对于模数为16的圆的另一种不同解释。将数值0到15用相应的二进制数系统中的4位向量0000, 0001, ..., 1111来表示。然后按照补码方式（见图2-1）重新将这些二进制向量解释成从-8到+7的有符号数，如图2-3b所示。

让我们将mod 16的加法技术应用在将+7加到-3的一个简单例子上。对于这两个数的补码表示分别是0111和1101。为了完成两数相加，在图2-3b的圆周上找到0111，然后按顺时针方向移动1101 (13) 步到达了0100，得到了+4的正确答案。如果用从右到左的按位加方法完成这个加法，我们得到：

$$\begin{array}{r} 0111 \\ +1101 \\ \hline 10100 \\ \uparrow \\ \text{进位} \end{array}$$

注意，如果在这个加法中忽略第四位上的进位，就得到了一个正确的答案。实际上就是这样做的。忽略这个进位是用 $N$ 取模算法的自然结果。当我们在图2-3b中绕着圆移动时，对于1111值的下一个值通常应该是10000。然后，又返回到了0000值。现在来陈述使用补码表示系统的 $n$ 位有符号数的加法和减法规则。

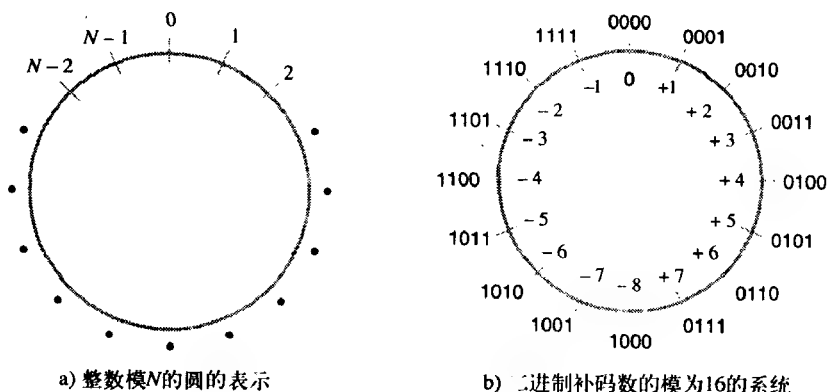


图2-3 模数系统和补码系统

1. 两个数相加，它们的 $n$ 个表示位相加，在最高有效位（MSB）上忽略进位符号。它们的和将是用补码表示的代数运算的正确值，只要结果是在 $-2^{n-1}$ 到 $+2^{n-1}-1$ 的范围之内。

2.  $X$ 和 $Y$ 两个数相减时，也就是执行 $X-Y$ 时，求出 $Y$ 的补码形式然后按照规则1将它加到 $X$ 中。同样，如果结果是在 $-2^{n-1}$ 到 $+2^{n-1}-1$ 的范围内，其结果将是用补码表示的代数运算的正确值。

在图2-4中给出了一些加法和减法的例子。在所有这些4位数的例子中，结果不会超过-8到+7的表示范围。当结果没有在表示范围之内时，便发生了算术溢出。这种情况将在下一节讨论。图2-4中(a)到(b)的四个加法运算符合规则1，(e)到(j)的六个减法运算符合规则2。减法运算要求减数（算式中下面的值）转换为补码形式，不论减数为正数或负数，用这样的变换操作其结果都是正确的。

(a)	0010	(+2)	(b)	0100	(+4)
	+ 0011	(+3)		+ 1010	(-6)
	0101	(+5)		1110	(-2)
(c)	1011	(-5)	(d)	0111	(+7)
	+ 1110	(-2)		+ 1101	(-3)
	1001	(-7)		0100	(+4)
(e)	1101	(-3)		1101	
	- 1001	(-7)		+ 0111	
				0100	(+4)
(f)	0010	(+2)		0010	
	- 0100	(+4)		+ 1100	
				1110	(-2)
(g)	0110	(+6)		0110	
	- 0011	(+3)		+ 1101	
				0011	(+3)
(h)	1001	(-7)		1001	
	- 1011	(-5)		+ 0101	
				1110	(-2)
(i)	1001	(-7)		1001	
	- 0001	(+1)		+ 1111	
				1000	(-8)
(j)	0010	(+2)		0010	
	- 1101	(-3)		+ 0011	
				0101	(+5)

图2-4 补码的加、减法操作

31

在补码系统中我们经常需要用比给定数要多的位数来表示一个数。对于一个正数，用左边加0的方法实现。对于负数，最左边的位是（符号位）等于1，具有同样值的较长位数的数可以用在左边根据需要多次重复符号位来实现。为了理解为什么这样做是正确的，考查图2-3b中的模为16的圆。将它与比较大的圆，即模为32或模为64的情况做比较，对于值-1、-2等的表示，它们完全相同，只是用若干个1加到了左边。概要地说，在补码系统中用一个大一些的位数表示一个有符号数时，按需要在左边重复它的符号位。这种操作称为符号扩展。

利用补码表示法简化了有符号数的加法或减法操作，这就是现代计算机采用补码表示法的理由。看起来似乎反码表示法与补码表示法一样有效，但是尽管数的求反很容易，可是加法运算以后生成的结果却不能保证总是正确的。进位 $c_n$ 不能被忽略。如果 $c_n = 0$ ，得到的结果是正确的，如果 $c_n = 1$ ，必须对这个结果加上1，以保证它的正确性。是否需要做这个修正以加法操作是否产生进位输出为条件的，这就意味着加法和减法运算在反码系统中不能像在补码系统中那样方便地实现。

#### 2.1.4 整数算术运算中的溢出

在使用补码表示数字的系统中， $n$ 位可表示的值在 $-2^{n-1}$ 到 $+2^{n-1}-1$ 之间。例如，使用4位可以表示数的范围是-8到+7，就像在图2-1中给出的那样。当算术运算的结果超出了这个表示范围时，就发生了一个算术溢出。

当无符号数相加时,最高有效位上的进位输出 $c_n$ 作为溢出标志。但是这种方法对于有符号数相加不起作用。比如,当使用4位有符号数时,如果我们将+7和+4相加,输出的和向量 $S$ 是1011,它是数值-5的代码,是一个不正确的结果。从MSB(最高有效位)位上得到的进位输出信号应该是0。类似地,如果我们将-4和-6相加,得到 $S = 0110 = +6$ ,又是一个错误的结果,因为在这种情况下进位输出信号应该为1。因此,如果两个加数有着同样的符号,就可能产生溢出。显然,使用不同符号的数做加法时不会产生溢出。这样得出了以下结论:

1. 只有当两个具有相同符号的数相加时才会产生溢出。
2. 当对有符号数做加法运算时,从符号位传来的进位输出信号不能充分说明发生了溢出。

检测溢出的一个简单方法是测试两个加数 $X$ 和 $Y$ 的符号,以及结果的符号。当两个操作数 $X$ 和 $Y$ 具有相同的符号时,而结果 $S$ 的符号与 $X$ 和 $Y$ 的符号不同时就发生了溢出。

### 2.1.5 字符

除了数字以外,计算机还必须能够处理由字符组成的非数字的文本信息。字符可以是字母、十进制数、标点符号等等。它们通常用8位长的代码来表示。最广泛使用的这种代码之一是在附录E中描述的ASCII码。

## 2.2 内存单元和寻址

数字和字符操作数,以及指令都存储在计算机的存储器中。现在我们考虑存储器是如何构成的。存储器是由几百万个存储单元构成的,其中每个单元可以存储一位具有0值或1值的信息。由于单独的一位只表示信息中一个非常小的量,所以很少单独对位做处理。常用的方法是按固定大小的组对位做处理。为了这个目的,对存储器进行构造以便于对 $n$ 位一组的数在单独的基本操作中进行存储或检索。每一个 $n$ 位组称为一个字, $n$ 称为字长。计算机的存储器可以用图表方式表示成字的集合,如图2-5所示。

现代计算机所具有的字长的一般范围是从16位到64位之间。如果一台计算机的字长是32位,那么如同图2-6所示,一个单独的字就能够储存一个32位的补码数或是四个各占据8位的ASCII码字符。8位的一个单元叫做一个字节。机器指令可能要用一个或多个字来表示。在描述了汇编语言级的指令后,在后面一节中将讨论机器指令是如何被编码到存储器的字中的。

为了储存或检索一个信息项去访问存储器,该信息项无论是一个字或是一个字节,对于每一项的位置要有具体的名字或是地址。习惯上我们用从0到 $2^k - 1$ 的数字( $k$ 取一些适当的值),作为存储器连续单元的地址。 $2^k$ 地址构成了计算机

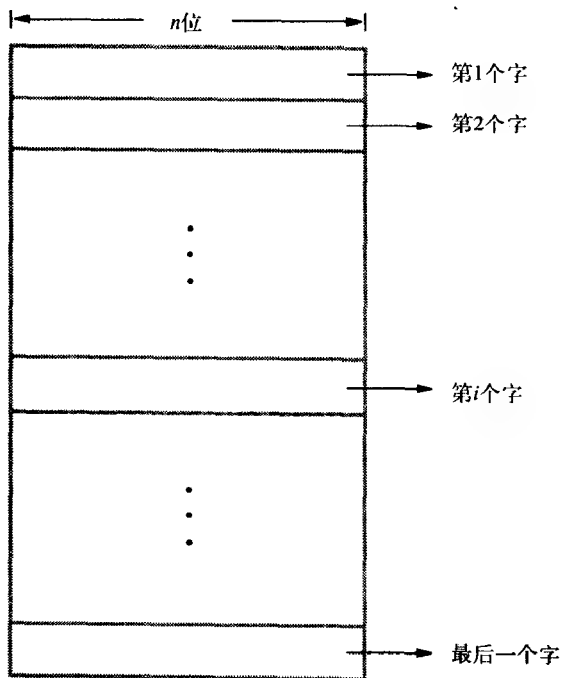


图2-5 存储器中的字

的地址空间，而存储器可以有高达 $2^k$ 的可寻址单元。例如，一个24位地址生成一个 $2^{24}$  (16 777 216) 存储单元地址空间。这个数通常写成16M，这里1M是 $2^{20}$  (1 048 576) 的数。一个32位的地址创建 $2^{32}$  或者4G的存储单元地址空间，这里1G表示为 $2^{30}$ 。其他的惯用标记法就是通常用K来表示数 $2^{10}$  (1 024)，用T表示数 $2^{40}$ 。

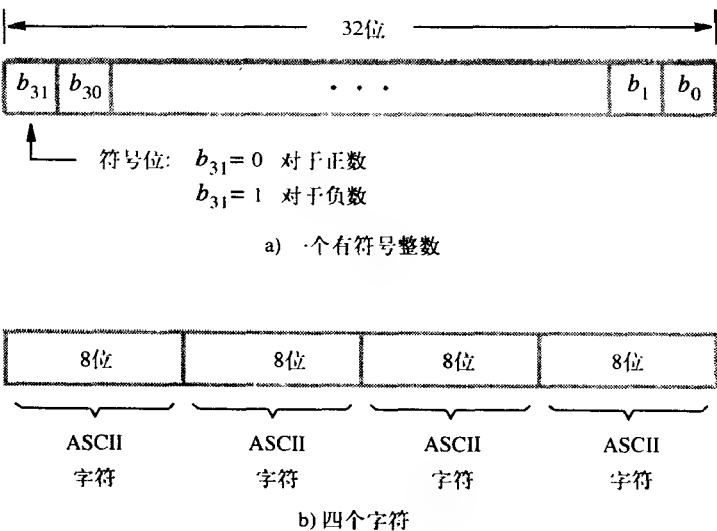


图2-6 在一个32位的字中编码信息的例子

2.2.1 按字节寻址能力

现在有三种基本的信息处理量：位、字节和字。一个字节通常为8位，但字长通常的范围是从16位到64位。为每一位分配不同的地址是不切实际的做法。大部分实际分配方法是将连续的地址对应于存储器中连续的字节单元。这是大多数现代计算机中使用的分配方法，也是我们在这本书中常用的方法。按字节寻址存储器就用于这种分配方法。字节单元具有地址0, 1, 2, …，这样，如果机器的字长为32位，那么连续的字被分配到地址0, 4, 8, …中，其中每个字包含四个字节。

33

2.2.2 big-endian和little-endian分配

就像在图2-7中给出的那样，为字分配地址有两种方法。当低字节地址作为一个字中的最高有效字节（最左边字节）时采用big-endian方法。而little-endian方法用于相反的次序中，在这里低字节地址作为一个字中最低有效的字节（最右边字节）。如同在2.1.1节中描述的那样，字中的“最高有效”和“最低有效”是指当这个字表示一个数时它们在分配位中所占的权重（以2为权）。Little-endian和big-endian两种分配法都在商业计算机中使用。在这两种情况中，都将字节地址0, 4, 8, … 作为存储器中连续字的地址和用来说明存储器读写操作中的地址。

34

在一个字中除了指明字节地址排序外，还需要说明每一位在一个字节或一个字中的标志。最常用的也是我们在这本书中采用的惯用方法在图2-6a中给出，它是数字型数据编码最常用的排序方法。这种排序法也适用于字节中位的标志，即从左到右是 $b_7, b_6, \dots, b_0$ 。但是，在有些计算机中使用的是相反的排序法。



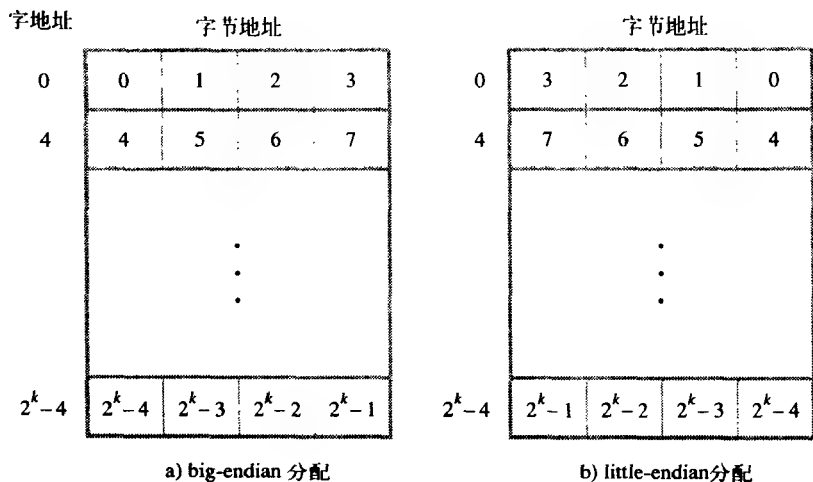


图2-7 字节和字的寻址

35

### 2.2.3 字的对齐

在32位字长的情况下，如图2-7所示，自然字的边界发生在地址0, 4, 8, ... 上。我们说这些字的位置具有对齐地址。一般地说，如果它们的开始处在一个字节地址上，这个地址又是在一个字的字节整倍数上，那么这些字在存储器中是对齐的。因为实际中会涉及到多种二进制码地址，一个字的字节数是2的幂次方，因此，如果字长是16（2字节），对齐字从字节地址0, 2, 4, ... 开始，而如果一个字长是64（2<sup>3</sup>字节），对齐字从字节地址0, 8, 16, ... 开始。

没有一个基本原则性的约定字不能从任意一个字节地址开始。如果一个字可以从任意的地址开始，这些字就是不对齐地址。虽然在大多数情况下使用的是对齐的地址，但有些计算机也允许使用不对齐地址的字。

### 2.2.4 访问数、字符和字符串

一个数通常占用一个字。在存储器中可以通过指明字的地址对其进行访问。同样，对于单个字符也可以通过它们的字节地址进行访问。

在许多的应用中，需要处理可变长度的字符串。字符串的起始点用串中包含的第一个字符的字节地址进行说明。后续的字节单元中包含着该串中连续的字符。有两种方法可以说明该串的长度，一个具有“串结尾”含义的特殊控制字符可以用来作为这个串的最后字符；另一种方法是使用一个单独的存储单元或是一个处理器寄存器，其中存放一个数字用来表示字符串的字节长度。

## 2.3 存储器操作

36

程序的指令和数据操作数都储存在存储器中。要执行一条指令，处理器控制电路必须要把包含这条指令的单个字（或多个字）从存储器传送到处理器中。操作数和操作结果也必须能够在存储器和处理器之间传送。因此，有关存储器的两个基本操作是必须的，即Load（或Read、Fetch）和Store（或Write）。

Load操作是指对一个指定存储单元中的内容做一个拷贝并将它传送到处理器中，存储器中

的内容保持不变。在开始Load操作时，处理器向存储器发送所期望得到单元的地址并且要求读取它的内容，存储器读出存储在那个单元中的数据并将它发送给处理器。

Store操作是从处理器向一个指定的存储单元中传送一条信息，它将破坏这个单元中原有的内容。处理器向存储器发送一个所要求的单元地址，同时带着将要写入这个单元中的数据。

任何一个字或一个字节的的信息项都能够在处理器和存储器之间使用单独的操作进行传递。就像在第1章中描述过的那样，处理器中包含有少量的寄存器，每个寄存器都能保存一个字。这些寄存器可以作为传送到存储器的源也可以作为从存储器中传送数据的目的地。当传送一个字节时，通常被放在寄存器的低位字节（最右边）上。

有关这些操作的硬件实现描述将在第5章和第7章中论述。在本章中我们使用的是ISA的观点，所以将目标集中在指令和操作数的逻辑处理上。具体的硬件部分，比如像对处理器寄存器的讨论仅限定在对机器指令和程序执行情况的理解范围内。

## 2.4 指令和指令序列

由一个计算机程序执行的任务是由一些小的执行步骤序列构成的，比如像两个数相加、测试特定的条件、从键盘上读一个字符或是发送一个字符到显示屏上去显示等。一台计算机必须具备能够执行以下四种类型操作指令的能力：

- 在存储器和处理器寄存器之间的数据传送
- 数据的算术和逻辑运算
- 程序序列和控制执行
- 输入/输出传送

我们首先讨论前两种指令类型。为了便于讨论，需要先介绍一些标记符号。

### 2.4.1 寄存器传送标记

我们需要描述的信息在计算机中从一个单元传送到另一个单元。在这种传送中可能涉及到的单元可以是存储器中的单元、处理器寄存器或是I/O系统中的寄存器。大多数情况下，用一个表示它的硬件二进制地址的符号名来识别其位置。例如，存储单元的地址名可能是LOC、PLACE、A、VAR2；处理器寄存器名可能是R0、R5；而I/O寄存器名可能是DATAIN、OUTSTATUS等。一个单元中的内容用这个单元的名字外加两个方括号来表示。因此表达式

$$R1 \leftarrow [LOC]$$

表示存储器单元LOC中的内容被传送到处理器寄存器R1中。

另一个例子是将寄存器R1和R2所包含地址的内容做加法操作，将它们的和放到寄存器R3中。这个动作表示为：

$$R3 \leftarrow [R1] + [R2]$$

这种标记方式就是所谓的寄存器传送标记（RTN）。要注意的是，在RTN表达式的右边总是表示一个值，而左边是存放这个值的单元名，将用这个值覆盖该单元中原有的内容。

### 2.4.2 汇编语言符号

我们需要用另一种标记符号表示机器指令和程序。为此，使用汇编语言形式。例如，一条

如上所述的产生传送操作的指令，将存储单元LOC的内容传送到处理器寄存器R1中，将使用这样的语句来说明：

Move LOC, R1

执行这条指令后LOC的内容不改变，但寄存器R1中原有的内容被覆盖了。

第二个例子处理器寄存器R1和R2中的两个数相加并将得到的和放到R3中，可以用汇编语言的语句描述为：

Add R1, R2, R3

### 2.4.3 基本指令类型

两个数相加的操作是任何一台计算机都能够完成的一种基本功能。语句：

$C = A + B$

在高级语言程序中是命令计算机去完成对A和B变量中的当前值做加法，并将得到的和放在第三个变量C中。当包含这个语句的程序被编译时，这三个变量A、B、C分配到了内存的不同单元中。我们将利用这些变量名去引用相应的内存单元地址。这些单元的内容表示这三个变量的值。因此，以上高级语言语句需要在计算机中完成的动作是：

$C \leftarrow [A] + [B]$

为了执行这个动作，存储单元A和B中的内容从存储器中被取出并且被传送到处理器中，在那里对它们进行求和计算。这个结果然后被传送回存储器中并存储在C单元中。

首先我们假设这个动作由单个机器指令完成，而且假定在这条指令中包含着三个操作数A、B、C的存储地址。这种三地址指令可以用符号表示为：

Add A, B, C

操作数A和B称为源操作数，C称为目的操作数，Add是将要对操作数执行的操作。这种类型的指令具有以下通用格式：

Operation   Source 1,   Source 2,   Destination  
(操作      源操作数1, 源操作数2, 目的操作数)

如果需要 $k$ 位说明每个操作数的内存地址，以上指令的编码形式除了需要说明Add操作位以外，还必须包含用于编址目的的 $3k$ 位。对于一个拥有32位地址空间的现代处理器来说，三地址指令放在一个字中对于一个普通字长显得太长了。因此，需要有一种允许在单个指令中使用多个字的格式来表示这种类型的一条指令。

一个可选择的方法是使用简单指令的一个序列去执行同样的任务，其中每条指令仅有一个或两个操作数。假定两地址指令格式：

Operation   Source,   Destination  
(操作      源操作数, 目的操作数)

是可以使用的。这种类型的Add指令就是：

Add A, B

它执行的操作是 $B \leftarrow [A] + [B]$ 。当这个和被计算出来时，结果送到存储器中并存放在B单元中，

替换了这个单元中原来的内容。这意味着操作数B既是源又是目的操作数。

一个单独的两地址指令不能解决我们原有的问题，原有的问题是对单元A和B的内容进行相加，并不破坏这两个单元中原有的内容，并且将和放到单元C中。这个问题可以用另一个将一个存储单元中的内容拷贝到另一个单元中的两地址指令来解决。这样的指令就是：

Move B, C

这里执行 $C \leftarrow [B]$ 的操作，保留单元B中的内容不变。操作“Move”在这里并不合适，它其实应该是“Copy”。但是，这个指令名已深深地被印在计算机术语表中了。 $C \leftarrow [A] + [B]$ 的操作现在可以用两条指令序列来完成：

Move B, C

Add A, C

39

在以上给出的所有指令中，首先指明的是源操作数，接着是目的操作数。这种顺序在许多计算机中被用于汇编语言对机器指令的表示中。但也有很多计算机中使用的源和目的操作数的顺序是相反的。在第3章中我们将会看到这两种表示法的例子。不幸的是，在这个问题上，没有一个明确的规定被所有的制造商所采用。事实上，即使是对于一台给定的计算机，它的汇编语言也可能对于不同的指令采用不同的顺序。在本章中，我们将继续使用源操作数在前的表达式进行描述。

我们已定义了三地址和两地址指令。但是，即便是两地址指令也许都不能被正确地放入到一个一般字长和地址大小表示的字中。还有一种可能是，有一些机器指令中只指明了一个内存操作数。当需要第二个操作数时，比如在一个Add指令中，它认为该操作数隐含在一个特定的单元中。一个叫做累加器的处理器寄存器可以用来完成此目的。因此，单地址指令：

Add A

的含义如下：将存储单元A中的内容加到累加寄存器中，然后将它们的和放回到累加寄存器中。我们再来介绍单地址指令

Load A

和

Store A

Load指令将存储单元A中的内容拷贝到累加器中，而Store指令是将累加器的内容拷贝到存储单元A中。只使用单地址指令， $C \leftarrow [A] + [B]$ 操作可以被以下指令序列来完成：

Load A

Add B

Store C

注意在指令中指明的操作数可以是源也可以是目的，具体依赖于这条指令的内容。在Load指令中，地址A指明的是源操作数，而目标单元即累加器是隐含的。另一方面，C在Store指令中代表着目标单元，而源即累加器是隐含的。

早期的一些计算机是围绕着一个单独的累加器结构设计的。大部分现代计算机设有大量的通用处理器寄存器——通常有8到32个，在有些情况下数目会更多。对这些寄存器中数据的访问比对存储在内存单元中数据的访问要快得多，因为寄存器是包含在处理器内部的。由于寄存器

40

的数量相对较少，只需要用少量的位数来指明是哪个寄存器参与了运算。例如对于32个寄存器来说，只需要5位地址。它比用给出一个在内存单元中的地址使用的位数要少得多。由于可以做到快速处理并且能缩短指令执行，寄存器常被用来存放处理器正在处理过程中的临时数据。

假设 $R_i$ 表示一个通用寄存器，指令：

Load A,  $R_i$   
Store  $R_i$ , A

和

Add A,  $R_i$

是对Load、Store和Add指令对于单独累加器情况的概括说明，这里寄存器 $R_i$ 完成的是累加器的功能。即便在这些情况下，当在一个指令中直接说明惟一的存储地址时，这条指令也有可能不能正常地放到一个字中。

当一个处理器有若干个通用寄存器时，许多指令所用到的惟一操作数被放在寄存器中。事实上，在许多现代处理器中，计算能够直接在处理器寄存器中保存的数据上执行。例如指令：

Add  $R_i$ ,  $R_j$

或

Add  $R_i$ ,  $R_j$ ,  $R_k$

都是这种类型的指令。在这两个指令中，源操作数是寄存器 $R_i$ 和 $R_j$ 的内容。在第一条指令中， $R_j$ 还当作目的寄存器，然而在第二条指令中，第三个寄存器 $R_k$ 当作目的寄存器。在这种指令中，因为只有寄存器的名字包含在指令中，所以可以被正常地放入到一个字中。

通常需要在两个不同的单元中传递数据，使用以下指令可以达到这一目的：

Move Source, Destination

它将源操作数中的内容拷贝到目标中。当数据传送到处理器的寄存器中或是从寄存器中移出时，可以使用Move指令而不是使用Load或Store指令，因为源操作数和目的操作数的顺序决定了所需要的是哪种操作。因此，

Move A,  $R_i$

相同于

Load A,  $R_i$

41 而

Move  $R_i$ , A

相同于

Store  $R_i$ , A

在本章中我们将用Move指令代替Load或Store命令。

对于那些允许算术运算可以将操作数只放在处理器寄存器中的处理器，作业 $C = A + B$ 可以用以下的指令序列完成：

Move A,  $R_i$   
Move B,  $R_j$

```
Add Ri, Rj
```

```
Move Rj, C
```

对于只有一个操作数在存储器中而其他的操作数必须在寄存器中的处理器，满足这个作业需要的指令序列应该为：

```
Move A, Ri
```

```
Add B, Ri
```

```
Move Ri, C
```

一个给定作业执行速度依赖于从存储器到处理器之间传递指令和传递这些指令需访问的相关操作数的时间。存储器之间传送要比在处理器内部传送慢得多。因此，当若干个运算被连续执行，而且它们的数据在处理器寄存器中，从而不需要将数据拷贝到存储器或从存储器中读取数据时，它的运算速度将会大大提高。当编译程序从高级语言生成机器语言程序时，控制数据在存储器和处理器寄存器之间来回传送的频率是十分重要的。

我们已讨论过三、两和单地址指令。但也有这种可能，指令中使用的所有操作数的位置都是隐含定义的。这种指令可以在那些将操作数存储在下推栈结构的机器中找到。在这种情况下，这种指令称为零地址指令。有关下推栈的概念将在2.8节中介绍，而使用这种方法的计算机将在第11章中讨论。

#### 2.4.4 指令执行和线性序列

在前面的指令格式讨论中，我们使用任务  $C \leftarrow [A] + [B]$  作了说明。图2-8给出了完成这个任务的程序段，当它存放在计算机存储器中的情况。我们已经假定该计算机允许每条指令有一个存放在存储器中的操作数并且该计算机有一定数量的处理器寄存器。假定该机字长为32位并且存储器是按字节可寻址的。程序中的三条指令放在连续的字单元中，起始单元是  $i$ 。以后的每条指

42

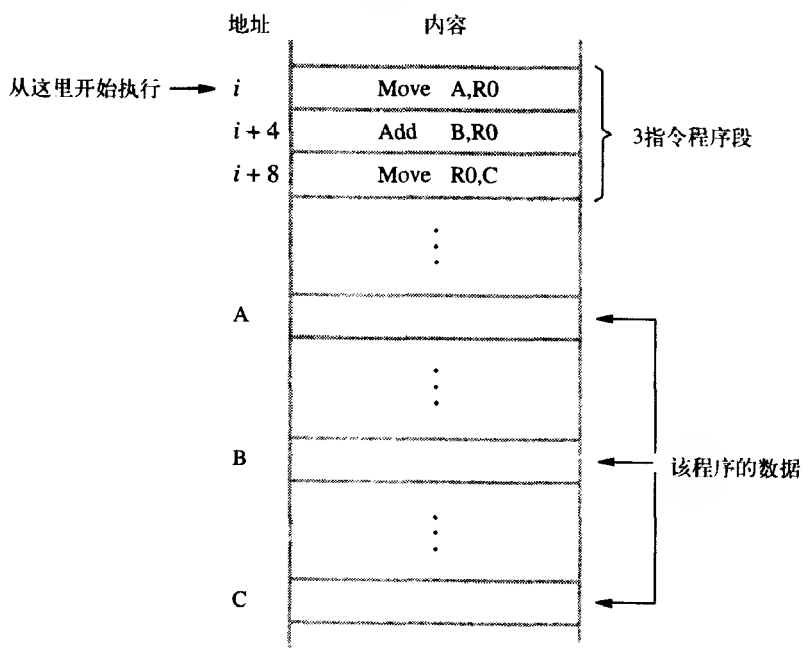


图2-8 一个完成  $C \leftarrow [A] + [B]$  的程序

令是4字节长,第二和第三条指令的起始地址为 $i+4$ 和 $i+8$ 。为了简单起见,还假定所有内存地址都能够在一个单字指令中直接说明,虽然这对于当前的处理器的地址空间范围和字长来说,通常是不可能的。

让我们来考虑这个程序将如何执行。处理器中包含一个称为程序计数器(PC)的寄存器,它保存着下一步将要执行指令的地址。当开始执行一个程序时,它的第一条指令地址(在本例中为 $i$ )必须放入PC中。然后处理器的控制电路利用PC中的这个信息,一次一个地按照递增地址的顺序获取并执行指令。这种执行方式叫做线性序列。在每条指令执行时,PC每次用4做递增值以指向下一条指令。这样当 $i+8$ 位置上的Move指令被执行后,PC中包含的值是 $i+12$ ,它是下一个程序段的第一条指令的地址。

43 执行一条给定指令可以分成两个阶段,第一个叫做取指令阶段中,根据PC提供的地址从存储单元中获取该指令。这条指令被放入到处理器的指令寄存器(IR)中。在第二个叫做指令执行的阶段开始时,对IR中的指令进行检查以确定将要执行哪种操作。这个被指明的操作然后被处理器执行。这个执行过程通常涉及到从存储器或处理器寄存器中提取操作数,执行一个算术或逻辑运算,然后将结果存放到目标单元中。在这两个过程中的某个点上,PC的内容被递增值使它指向下一条指令。当一条指令的执行阶段完成时,PC中包含着下一条指令的地址,并且一个新的读取指令阶段又可以开始了。在大多数处理器中,执行阶段本身又被划分成了一些相应的取操作数、执行运算以及储存结果的不同阶段。

#### 2.4.5 转移

考虑对一个有 $n$ 个数的列表相加的任务。图2-9中的程序轮廓是对图2-8中程序的一个概括。包含有 $n$ 个数的存储单元的地址用符号NUM1、NUM2、...、NUM $n$ 给出,并且使用一个单独的Add指令将每个数累加到寄存器R0中,当所有的数加完以后,结果被存放在存储单元SUM中。

44 代替使用一长串的Add指令,可将一个单独的Add指令放在一个循环程序中,就像在图2-10中看到的那样。这个循环是一个顺序指令序列,根据需要多次被执行。它的开始点在LOOP单元处,结束点在指令Branch >0处。每经过一轮这个循环时,列表中下一个元素的地址就被确定下来,然后这个元素被提取出来并加到R0中。操作数的地址可以用多种方式指出,这些将在2.5节中介绍。这里,我们把注意力集中在如何创建并控制程序的循环上。

假设这个列表中元素的数量为 $n$ , $n$ 被储存在内存单元N中,如下图所示。寄存器R1当作一个计数器,确定执行循环的次数。因此,单元N中的内容在程序开始时被装入到寄存器R1中。然后在循环体内部,指令

Decrement R1

45 每当通过一次该循环时对R1的内容减1(类似的这种操作是执行一条增值指令,它对操作数每次加1)。只要递减操作的结果是大于0的,就重复执行这个循环。

现在我们来介绍转移指令。这类指令将一个新的值加载到程序计数器中,因此,处理器按这个新地址(该地址称为转移目标)获取并执行这个指令,替代了按顺序紧挨着转移指令的那个地址单元中的指令执行。条件转移指令只有在给出的条件满足时才产生一个转移。如果条件不满足,PC按正常的方式进行增值,连续地址顺序的下一条指令被提取并被执行。

在图2-10的程序中,指令

Branch >0 LOOP

(如果大于0就转移)是一个条件转移指令,如果前一条指令的执行结果是大于0的,也就是寄存器R1中的递减值是大于0的,就产生一个转去执行LOOP单元中指令的转移指令。这意味着只要在列表中有元素还可以被加到R0中,这个循环是要重复执行的。在 $n$ 遍循环过后,递减指令产生了一个0值,因此转移不再发生。替代它的是Move指令被提取并被执行。它将最后的结果从R0中移到内存单元SUM中。

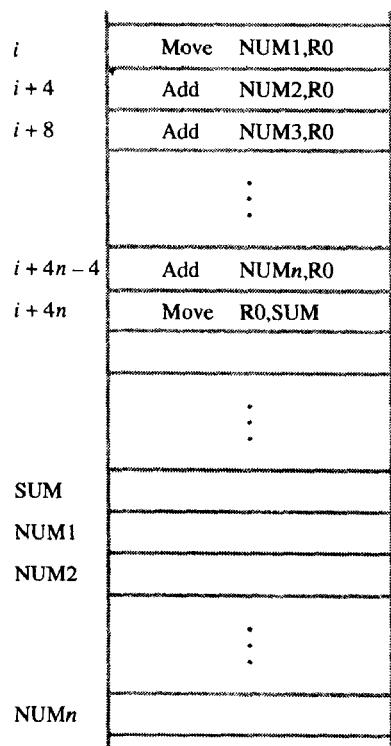


图2-9 一个用于累加 $n$ 个数的顺序程序

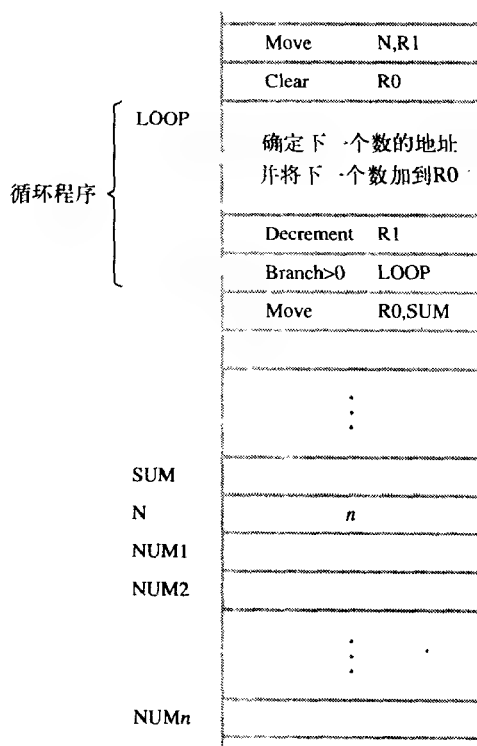


图2-10 用一个循环完成 $n$ 个数相加

这种首先测试条件然后从设置的选择中选择一种方式继续执行的计算能力比仅仅完成循环控制的方式有着更多的应用价值。这种能力在所有计算机指令系统中都可以找到,并且它对于大多数重要任务的程序设计也是必须的。

## 2.4.6 条件码

为了随后的条件转移指令的使用,处理器要保存有关各种操作执行结果的过程信息。为了达到这一要求可以使用在单个位上记录所需要信息的方法,通常这些位称为条件码标志。这些标志通常会被聚集在一起形成一个特殊的处理器寄存器,该寄存器叫做条件码寄存器或是状态寄存器。每个单独的条件码标志可以根据操作执行结果,被设置为1或清除成0。

四种常用的标志是:

N (负数) 如果结果是负数置成1; 否则清除为0

Z (零) 如果结果是0置成1; 否则清除为0

V (溢出) 如果运算发生溢出置成1; 否则清除为0

C (进位) 如果运算结果有一个进位输出置成1; 否则清除为0



N和Z标志表示算术或逻辑运算的结果是否为负数或0。N和Z标志还会受传递数据指令的影响,例如像Move、Load或Store这样的指令。它有可能对以后的条件转移指令产生影响,可以基于被传送的操作数的符号和值去产生一个转移。有些计算机还提供一条特殊的Test指令,它检查某个寄存器或某个存储器中的值,然后根据检查情况置位或清除N和Z的相应标志。

V标志表示溢出是否发生。就像在2.1.4节中解释的那样,当一个算术运算的结果超出了操作数可使用位数能够表示值的范围时,就产生了溢出。处理器对V标志进行置位,使程序员可以去测试是否已经发生了溢出并能够转移到这个问题的正确执行的程序段中。像BranchIfOverflow这样的指令就是为这一目的提供的。还有,就像我们将在第4章中看到的那样,V位被置位可能会引起一个程序自动产生中断,并且由操作系统决定下一步将要做什么。

如果在算术运算中从最高有效位产生一个进位,C标志就被置成1。这个标志使得在执行算术运算中的操作数可以比处理器的有效字长还要长。这种操作可以用在多精度的算术运算中,这些将在第6章讨论。

在2.4.5节中讨论的指令Branch > 0是一个测试单个或多个条件标志的转移指令例子。如果被测试的值既不为负数又不等于零,就产生一个转移。就是说,如果N和Z都不为1,就产生一个转移。还会提供许多其他的条件转移指令以满足各种条件的测试。这些条件是作为与条件标志码有关的逻辑表达式给出的。

在一些计算机中,条件码标志自动地受执行算术或逻辑运算指令的影响。然而,这并不是普遍的情况。例如,许多计算机中有两个版本的Add指令。一种版本Add不影响标志位;而另一种版本AddSetCC就对标志位造成影响。这给程序员和编译程序在对程序做流水线执行的预处理时提供了更多的灵活性,流水线执行我们将在第8章中讨论。

### 2.4.7 生成存储器地址

让我们回到图2-10中,LOOP指令块的目的是在每次经过循环时,从列表中加一个不同的数。因此,在每次循环中,这个块中的Add指令必须访问一个不同的地址。怎样来指明这个地址呢?内存操作数地址无法在循环中用一个单独的Add指令直接给出。否则,它将需要在每次的循环中进行修正。作为一种可能情况,假设处理器寄存器R<sub>i</sub>用来作为保存一个操作数的内存地址。如果在循环之前,R<sub>i</sub>用NUM1作为初始装载地址,以后每通过一次循环后用4进行递增,就可以提供所需要的地址了。

在这种以及其他很多类似的情况下,更多的是根据需要用灵活方式去指明一个操作数的地址。一个计算机的指令系统通常提供了多种这样的方法,它们称为寻址方式。尽管计算机之间的具体情况不同,但是其基本概念是相同的。我们在下一节中将讨论这些问题。

## 2.5 寻址方式

现在我们已经看到了一些汇编语言程序的简单例子。一般来说,一个程序是对储存在计算机存储器中的数据进行操作的,这些数据可以用多种方式组织。如果想要保留学生姓名,我们可以将它们写在一个表中。如果想将每个名字与一些信息联系起来,比如记录电话号码或各门课程的分数,可以用表格的形式构成这些信息。程序员使用叫做数据结构的组织结构表示用于计算中的数据。这些数据结构包括表、链表、数组、队列等。

程序通常是用高级语言编写的，在高级语言中程序员可以使用常数、局部和全局变量、指针和数组。当把一个高级语言程序翻译成汇编语言时，编译程序必须能够使这些结构有效地利用一些便利条件，而这些便利条件是在将要运行该程序的计算机指令系统中已经提供的。在一条指令中具体指明操作数位置的不同方法称为寻址方式。在这一节中，我们给出现代处理器中最主要的寻址方式，在表2-1中给出了一个简要的描述。

表2-1 通用的寻址方式

名 称	汇 编 语 法	寻 址 功 能
立即	#Value	操作数 = Value
寄存器	Ri	EA = Ri
绝对 (直接)	LOC	EA = LOC
间接	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
变址	X (Ri)	EA = [Ri] + X
带变址基址	(Ri, Rj)	EA = [Ri] + [Rj]
带变址和偏移量的基址	X (Ri, Rj)	EA = [Ri] + [Rj]+X
相对	X (PC)	EA = [PC] + X
自动递增	(Ri)+	EA = [Ri] 递增Ri
自动递减	· (Ri)	递减Ri EA=[Ri]

48

EA = 有效地址  
Value = 有符号数

2.5.1 变量和常数的实现

变量和常数是最简单的数据类型，并且几乎在任何一个计算机程序中都会存在。在汇编语言中，一个变量用分配保存该变量值的寄存器或是内存单元来表示。因此，这个值可以根据需要使用适当的指令来改变。

在2.4节的程序中只使用了两种寻址方式去访问变量。我们通过指明寄存器的名称或是操作数被装入的内存地址单元地址来访问操作数。这两种方式的具体定义是：

寄存器方式——操作数是处理器寄存器中的内容；在指令中给出寄存器的名称（地址）。

绝对方式——操作数在一个存储器单元中；指令中明确地给出了这个单元的地址（在有些汇编语言中，这种方式叫做直接方式）。

指令

```
Move LOC, R2
```

使用了这两种方式。处理器寄存器当作临时储存单元，这时在寄存器中的数据使用寄存器方式进行访问。绝对方式在程序中可以表示全局变量。例如在高级语言程序中有这样一个声明：

```
Integer A, B
```

它将导致编译程序为变量A和B分别分配一个内存单元。每当它们在以后的程序中被引用时，编译程序就产生绝对方式去访问这些变量的汇编语言指令。

下面让我们来考虑常数的表示方法。地址和数据常数在汇编语言中可以用立即方式来表示。

立即方式——操作数在指令中被明确地给出。



包含一个操作数地址的寄存器或内存单元称为指针。间接方式和指针的使用在程序设计中是一个重要和强有力的概念。考虑一个类似于寻找宝物的问题：在寻找指令中可能告诉你要去寻找一个房子的地址，而不是所要寻找的宝物的地址；在房子中你找到了一个提示信息，它给了你另一个地址，在这个地址中你将会找到宝物。随着提示信息的变化，宝物的位置可以被改变，但是寻找宝物的指令是相同的。改变提示信息等于在计算机程序中改变一个指针的内容。例如，在图2-11中随着寄存器R1或单元A中内容的改变，同样的Add指令取到了不同的操作数，并将它们加到寄存器R0中。

现在让我们回到图2-10对于一个列表中的数字做加法的程序中。间接寻址可以用来访问表中连续的数据，程序的修改结果在图2-12的程序中给出。寄存器R2被当作列表中数字的指针，操作数通过R2间接地被访问。程序的初始化部分从存储单元N中将计数器的值加载到R1中，并且使用立即寻址方式将地址值NUM1放置到R2中，这个地址是列表中第一个数的地址。然后将R0的内容清为0。图2-12中循环体里的前两条指令执行以LOOP开始的未明确指明的指令块，如图2-10所示。第一次通过循环时，指令

```
Add (R2),R0
```

从单元NUM1获取操作数，并把它加到R0中。第二条Add指令将指针R2的内容加上4，这样在通过第二遍循环执行以上指令时，R2中将包含着地址值NUM2。再来看C语言语句

```
A = * B
```

这里B是一个指针变量。这个语句可以被编译成：

```
Move B, R1
Move (R1), A
```

51

通过存储器使用间接寻址，用下面的指令能达到同样的效果：

```
Move (B), A
```

尽管表面上看很简单，通过存储器的间接寻址方式已经被证明是一种用途有限的寻址方式，在现代计算机中已不常见了。在第8章中我们将看到这种指令，它需要两次访问存储器去得到一个操作数，不能很好地适应流水线技术的执行。

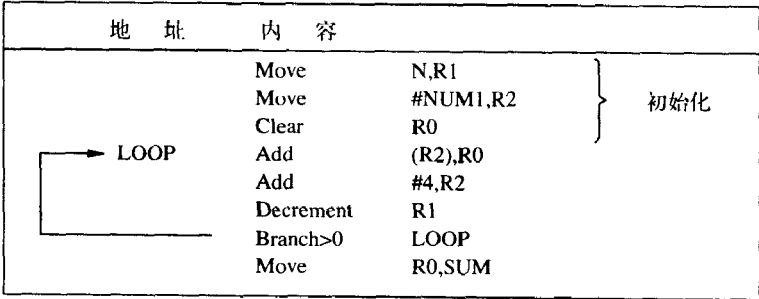


图2-12 在图2-10中的程序中使用间接寻址方式

通过寄存器进行间接寻址已经被广泛应用。图2-12中的程序展示了它所提供的灵活性。另外，当绝对寻址无法使用时，通过寄存器的间接寻址方式使它能够用第一个装入到寄存器中的操作数地址去访问全局变量。

### 2.5.3 变址和数组

下一个我们将要讨论的寻址方式提供了访问操作数不同方式的灵活性。它对于列表和数组的处理很有用。

变址方式——操作数有效地址是由对一个寄存器中的内容加上一个常数值而生成的。

使用的寄存器可以是专门为这一目标设立的专用寄存器，更通用的方式是，它可以是处理器中设置的任何一个通用寄存器。无论是哪种情况，它都被称为变址寄存器。我们表示变址方式的符号是：

$$X(Ri)$$

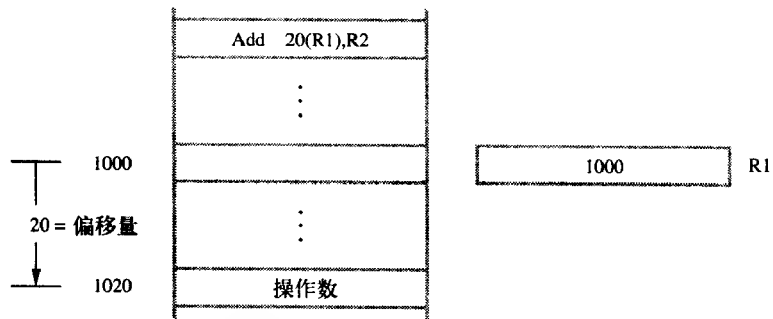
这里X表示在指令中保存的常数值，而Ri是相关寄存器的名字。该操作数的有效地址用下面的式子给出：

$$EA = X + [Ri]$$

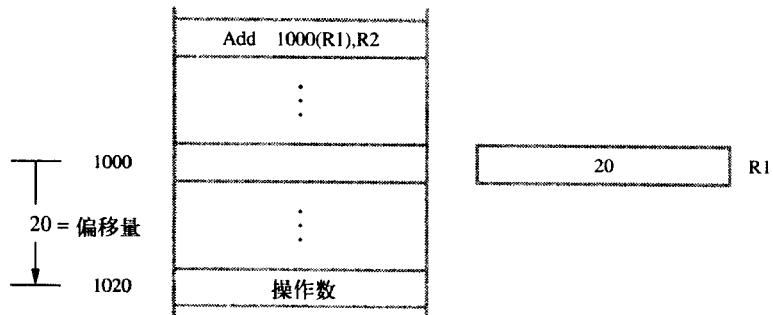
变址寄存器的内容在生成有效地址的过程中不能被改变。

在汇编语言程序中，常数X可以用直接说明的数字或用表示数字值的符号名给出。在这种方法中一个符号名是与一个特定的数字值相关的，这将在2.6节中讨论。当指令被转换成机器码时，常数X作为指令的一部分被给出并且通常用比计算机的字长要少的位数来表示。因为X是一个有符号的整数，在将它与寄存器中的内容相加之前，必须扩展它的符号位使其与寄存器的字长相同（请看2.1.3节）。

图2-13说明了两种使用变址的方法。在图2-13a中，指针寄存器R1中包含有一个存储单元的



a) 偏移量作为一个常数给出



b) 偏移量在变址寄存器中

图2-13 变址寻址

地址, 数值X定义了一个从这个地址到找到操作数位置的偏移量 (也称为位移量)。在图2-13b中具体说明了所使用的另一种选择。这里, 常数X相当于一个内存地址, 而这个变址寄存器中的内容定义了一个到操作数的偏移量。无论在何种情况下, 有效地址是两个值的和; 一个是在指令中明确给出的, 另一个是存储在一个寄存器中的。

为了理解变址寻址的作用, 考虑一个有关学生选修课程考试分数表的一个简单例子。假设这个成绩表开始的位置在LIST处, 它的结构如图2-14所示。4个字长的存储块构成一个记录, 记录中储存着与每个学生相关的信息。每个记录由学生的标识号 (ID), 紧接着是学生在三个测验中所得的分数构成。在这个班里有 $n$ 个学生, 这个 $n$ 值被存储在单元N中, 这个单元紧挨在这张表的前面。按照学生的ID号和考试分数给出信息, 假设该存储器是按字节可编址的并且字长是32位的。

我们应该注意到, 图2-14中的这张表代表着一个 $n$ 行4列的二维数组。每行包含一个学生的数据项, 而每列给出了ID号和一个考试分数。

假设我们想要计算在每次考试中所有得分的总和, 并将这三个和储存在存储单元SUM1、SUM2和SUM3中。图2-15中给出了一个可以用于这项任务的程序。在循环体内, 程序按照图2-13a中描述的方式使用了变址寻址方式, 去访问一个学生记录中三个得分中的每一个分数。寄存器R0当作为变址寄存器。在进入循环前, R0被设置成指向第一个学生记录ID的单元; 因此, 它包含的是LIST的地址。

在通过第一次循环时, 第一个学生的考试得分加到了保存在寄存器R1、R2和R3的和, 这些值在初始化时清成了0。这些得分可以使用变址寻址方式4(R0)、8(R0)和12(R0)进行访问。变址寄存器R0用16进行增值, 指向第二个学生的ID单元。寄存器R4初始化时的内容为 $n$ 值, 每一次循环结束时它的内容减1。当R4的内容减到0时, 所有学生的记录就都被访问到了, 并且循环结束。

在那之前, 条件转移指令将控制返回到循环的开始处去处理下一个记录。最后三条指令将R1、R2和R3中的累加和传递到相应的内存单元SUM1、SUM2和SUM3中。

这里要强调的是变址寄存器R0中的内容, 当它作为访问所得分数的变址寻址方式时其值不能够被改变。R0的内容只能被循环中的最后一个Add指令改变, 这样从一个学生的记录转向下一个学生的记录。

一般地说, 变址方式有助于访问那些特定的操作数, 这些操作数的存储单元是用相对于保

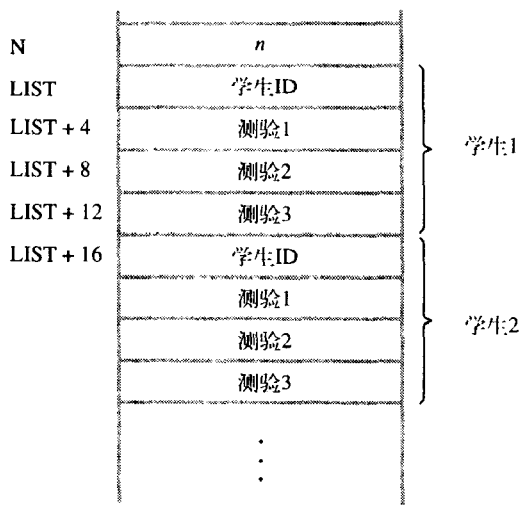


图2-14 一张学生记分表

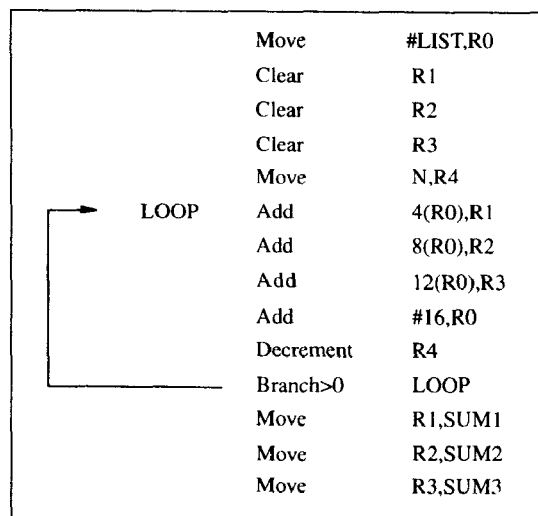


图2-15 用于访问图2-14中给出的  
考试分数表的变址寻址方式

存操作数的数据结构中的一个参照点来定义的。在刚刚给出的这个例子中，学生连续记录的ID位置是参考点，考试分数是使用变址寻址方式访问的操作数。

我们已经介绍了变址寻址的最基本的形式。在实际编程中还有一些这类基本形式的变形，它们对内存操作数的访问也是非常有效的。例如，第二个寄存器可以用来作为保存偏移量X，在这种情况下我们可以将变址方式写成：

$$(R_i, R_j)$$

其有效地址是寄存器 $R_i$ 和 $R_j$ 内容的和。第二个寄存器通常称做基址寄存器。这种变址寻址方式在访问操作数方面提供了更多的灵活性，因为，有效地址的两部分都可以被改变。

作为应用这些灵活性的一个例子，我们再来考虑图2-14中给出的学生记录的数据结构。在图2-15的程序中，在三个Add指令循环的开始处使用了不同的变址值去访问不同的考试分数。假设每个记录中包含有许多的数据项，可能比在这个例子中给出的三个考试分数项要多得多。在这种情况下，我们就需要用一条包含有第二个循环（嵌套）的指令来替换这三条Add指令。也就是这个记录连续单元的起始位置（参考点）被保留在指针寄存器 $R_0$ 中，相对于寄存器 $R_0$ 中的内容到各个项的偏移量可以保存在另一个的寄存器中。这个寄存器的内容每当通过一次内部循环时将被增值（请看习题2.9）。

变址方式还有一种使用两个寄存器加上一个常数的版本，这可以表示为：

55

$$X(R_i, R_j)$$

在这种情况下，有效地址是常数X与寄存器 $R_i$ 及 $R_j$ 内容的和。这种新增加的灵活性在访问一条记录内部的每一项中多个部分时是有用的，这里一个项的开始是由寻址方式的 $(R_i, R_j)$ 部分来指明的。换句话说，这种方式实现了一个三维数组方式。

## 2.5.4 相对寻址

我们已经用处理器的通用寄存器定义了变址方式。如果程序计数器PC是利用一个通用寄存器代替的，就得到了这种方式的另一个有用版本。这时 $X(PC)$ 可以当作一个内存单元的地址，它离程序计数器指向的位置有X个字节的距离。以后的地址单元是“相对”于程序计数器而确定的，而程序计数器总是指出一个程序中的当前执行位置，相对的模式名字与这个寻址类型有关。

**相对模式**——有效地址使用变址方式来确定，在此变址方式中用程序计数器替代通用寄存器 $R_i$ 。

这种方式可以用于访问数据操作数。但是，它的最常用用途是在转移指令中指明目标地址。比如有这样一条指令：

Branch > 0 LOOP

如果转移条件满足，可使得程序执行转移到由名字LOOP命名的目标单元上。这个单元的位置可以用所说明的一个从程序计数器的当前值开始的偏移量计算而得到。由于转移的目标可能在转移指令的前面或后面，所以给出的偏移量是一个有符号的数。

回想一下在一条指令执行的过程中，处理器对PC做增值使其指向下一条指令。大多数计算机在相对模式中计算有效地址时使用这个更新的值。例如，假设在图2-12的程序中Branch指令使用相对模式生成转移目标地址LOOP。假定该循环体中的四条指令，从LOOP处开始分别位于内存单元1000、1004、1008和1012中。因此，生成目标地址时PC的最新内容将是1016。为了转向

单元LOOP(1000)，需要的偏移值是 $X = -16$ 。

汇编语言中允许转移指令使用标号写出转移的目标，如图2-12所示。当汇编程序处理这样的指令时，它计算需要的偏移值，在这种情况下该值是 $-16$ ，并且使用寻址方式 $-16(PC)$ 产生相应的机器指令。

2.5.5 附加方式

到目前为止已讨论了五种在大多数计算机中可见到的基本寻址方式——立即、寄存器、绝对（直接）、间接和变址方式。我们还给出了许多变址方式的通用版本，所有这些寻址方式可能并不能在某一台计算机中都找到。尽管这些方式对于一般的计算是足够了，但是许多计算机还提供了附加模式，其目的在于协助某些特殊程序的设计任务。下面描述的两种方式对于访问内存的连续存储单元中的数据项是有效的。

56

自动递增方式——该操作数的有效地址是在指令中指明的一个寄存器中的内容。每当访问过该操作数后，这个寄存器的内容自动增加，去指向某个表中的下一个数据项。

我们采用将指明的寄存器放在圆括号中的方法来表示这种自动递增方式，表示这个寄存器的内容是作为有效地址使用的，后面跟着一个加号表示这些内容在操作数被访问后将要被增值。因此，自动递增方式被写成：

$(Ri) +$

默认情况下，当按这种方式给出时其递增量是1。但在一个按字节寻址的存储器中，这种方式将仅对访问一些表的连续字节有用。为了在一个按字节寻址的具有32位字长的存储器中访问连续的字，这个增量必须是4。对于具有自动递增方式的计算机，它可以用与访问操作数的大小相符的值来递增寄存器中的内容。因此，对于与字节大小相同的操作数其增量是1，对于16位的操作数其增量是2，对于32位的操作数其增量是4。因为操作数的大小通常是作为一个指令操作代码的一部分被说明的，所以用 $(Ri) +$ 来表示自动递增方式就足够了。

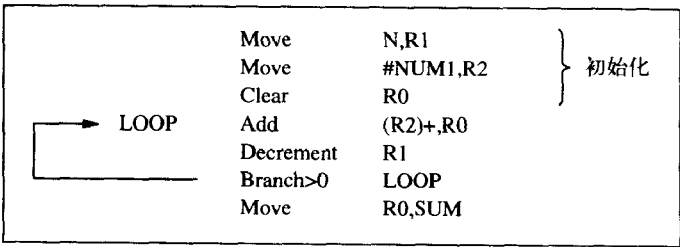
如果可以使用自动递增方式，可以将它用在图2-12中的第一条Add指令中，而第二条Add指令可以被取消。这个改进后的程序在图2-16中给出。

作为自动递增方式的一种匹配方式，另一种访问逆序列表数据项的有效方式是：

自动递减方式——指令中给出的寄存器的内容首先做自动递减，然后作为这个操作数的有效地址使用。

我们采用将这个指明的寄存器放在圆括号中的形式来表示自动递减方式，前面用一个减号说明在作为一个有效地址使用之前，这个寄存器中的内容要被递减。因此，我们写做：

$-(Ri)$



57

图2-16 对图2-12中的程序使用自动递增寻址方式



在这种方式中，操作数按照递减地址顺序进行访问。读者一定会奇怪，为什么在使用自动递减方式中地址使用之前要做递减，而在使用自动递增方式中在地址使用之后才做递增。这其中的主要理由在2.8节中给出，在那里我们将说明如何将这两种方式一起使用来实现重要的堆栈数据结构。

自动递增和自动递减寻址方式执行的动作显然可以用两条指令来完成，一条是访问操作数，另一条是对保存操作数地址的寄存器做增值或减值。将两种操作合并成一条指令，从而可以减少执行这个给定任务所需的指令数量。但是，在第8章中我们将指出，将两种操作合并在一个单指令中并不总是有利的。

## 2.6 汇编语言

机器指令用0和1模式来表示。这种模式在讨论或准备程序时是很不便的。因此，我们使用符号名来表示这些模式。到目前为止，已经使用了自然单词比如像Move、Add、Increment以及Branch作为指令操作去表示相应的二进制码模式。当为一台指定的计算机写程序时，这些单词通常使用称为助记符的缩写形式来代替，比如，MOV、ADD、INC和BR。同样地，我们使用标识符R3来表示寄存器3，用LOC表示一个存储器单元。这些符号名以及使用规则的一个完整集合构成了一种程序设计语言，通常叫做汇编语言。使用这些助记符描述完整的指令和程序的规则集合称为这种语言的语法。

使用汇编语言编写的程序可以被叫做汇编程序的程序自动翻译成机器指令的序列。这个汇编程序是实用程序集中的一个，它是系统软件的一部分。汇编程序与任何其他程序一样，作为机器指令的一个序列被存储在计算机的内存中。一个用户程序通常是通过键盘输入到计算机中，并储存在内存或是磁盘的任意位置上的。在这点上，用户程序简单地被看作为字符数字流的一个集合。当汇编程序执行时，它读取用户程序做分析，然后生成所需要的机器语言程序。而机器语言中包含着将要被计算机执行的用0和1模式说明的指令。按照原始的字符数字文本形式组成的用户程序叫做源程序，汇编后的机器语言程序叫做目标程序。我们将在2.6.2节中讨论汇编程序是如何工作的。首先给出一些汇编语言本身的外部特征。

汇编语言对于一个给定的计算机来讲可能区分也可能不区分大小写，也就是说，它可以区别也可以不区别大写和小写字母。为了改善文本的易读性，在例子里我们用大写字母表示所有名称和标号。例如，将Move指令写成：

MOVE R0, SUM

助记符MOVE表示被这个指令执行操作的二进制模式，或叫做OP码。汇编程序将这个助记符翻译成这台计算机能够理解的二进制OP码。

OP码助记符后边最少跟随一个空格字符，再后面的信息给出了具体的操作数。在我们的例子中，源操作数是在寄存器R0中。这个信息后面跟随的是目的操作数的具体说明，用一个逗号与源操作数隔开，中间不插入空格。目的操作数在内存单元中，它使用名字SUM来表示它的二进制地址。

因为对于具体操作数的单元可以有多种寻址方式来说明，汇编语言必须说明使用的是哪一种方式。例如，一个数字的值或者一个单独使用的名字，就像前面指令中的SUM，可以用来表示绝对方式。#号通常表示一个立即操作数。这样指令

ADD #5, R3

表示对寄存器R3的内容加上数字5并且将结果放回到寄存器R3中。这个#号不是惟一表示立即寻址的方法。在有些汇编语言中，所需要的寻址方式在OP码的助记符中指出。在这种情况下，一个给定的指令对于不同的寻址方式有着不同的OP码助记符。例如，前面的Add指令可以写成

```
ADDI 5,R3
```

助记符ADDI中的后缀I表示源操作数是用立即寻址方式给出的。

间接寻址通常由放置在圆括号中的代表指向操作数的指针名或符号名来表示。例如，如果数字5需要放置在一个存储单元中，它的地址被保存在寄存器R2中，期望的动作可以这样表示：

```
MOVE #5,(R2)
```

或者是：

```
MOVEI 5,(R2)
```

2.6.1 汇编指示

除了对在程序中表示的指令提供一种机制以外，汇编语言还允许程序设计员说明将源程序翻译成目标程序时所需要的其他信息。我们已提到过需要对任何一个在程序中使用的名字分配数字的值。假设名字SUM用来表示值200，这个情况可以通过这样一条语句传递到汇编程序的系统中：

```
SUM EQU 200
```

这个语句在目标程序运行时不代表一条将要被执行的指令；实际上，它甚至不出现在目标程序中。它只是简单地通知汇编程序，名字SUM无论出现在程序的任何地方都要用值200来取代。这样的语句叫做汇编指示（或汇编命令），它在汇编程序将源程序翻译成目标程序时起作用。

为了进一步说明汇编语言的使用，让我们重新考虑图2-12中的程序。为了在计算机中运行这个程序，需要按照汇编语言的要求编写源代码，具体说明生成目标程序所有需要的信息。假设它的每条指令和每个数据项占据内存中的一个字，这是一种非常简单的情况，但有助于保持这个例子的简单性。另外假设存储器是按字节寻址的并且它的字长是32位。还假定这个目标程序Load到了主存储器中，如图2-17所示。图中给出了该程序为了运行被Load后的机器指令和所需数据项占据的内存地址。如果汇编程序按这种分配产生目标程序，它需要知道：

- 怎样去解释名字。
- 在内存的什么位置存放指令。
- 在内存的什么位置存放数据操作数。

为了提供这些信息，源程序可以写成如图2-18给出的形式。这个程序以汇编指示开始，我们已讨论了Equate指示符EQU，它通知汇编程序SUM的值是多少。第二个汇编指示ORIGIN告诉汇编程序在内存的什么位

LOOP

```
SUM 200
N 204
NUM1 208
NUM2 212
NUMn 604
```

100	Move	N,R1
104	Move	#NUM1,R2
108	Clear	R0
112	Add	(R2),R0
116	Add	#4,R2
120	Decrement	R1
124	Branch>0	LOOP
128	Move	R0,SUM
132		
		⋮
		100
		⋮

图2-17 图2-12程序的内存分配

置存放以下的数据块。在本例情况下，被指明的地址单元是204。因为这个单元装入了数值100（它是这个列表中的项数），需要用DATAWORD指示符告诉汇编程序这个要求。它说明数据值100将被存放在地址为204的内存字中。

	内存 地址 标号	操作	地址 或 数据信息
汇编指示	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
		RESERVE	400
生成机器指令 的语句	START	ORIGIN	100
		MOVE	N,R1
		MOVE	#NUM1,R2
	LOOP	CLR	R0
		ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
		RETURN	
汇编指示		END	START

图2-18 对于图2-17中的程序使用汇编语言表述的形式

任何将指令或数据存放在内存单元的指令，可以给出一个内存地址标号。这个标号被分配了一个值，这个值等于那个单元的地址。因为，DATAWORD语句给出的标号是N，名字N被分配的值是204。无论何时在程序的其他任何地方碰到N，它都被替换成这个值。这种将N当作一个标号的方式等价于使用汇编指令：

N EQU 204

RESERVE指示符宣布这400个字节的内存块是专门用来保存数据的，并且名字NUM1与地址208相关联。这个指令并不能完成将任何数据加载到这些单元里，只有使用输入程序，才能把数据装入到该内存中，就像我们要在本章后面解释的那样。

第二个ORIGIN指示符说明目标程序的指令将从内存起始地址100开始Load。它后面跟着的是使用相应的助记符和语法编写的源程序指令。源程序最后一条语句是汇编指示END，它告诉汇编程序这里是源程序正文的结束点。END指令中包含标号START，它是这个程序开始执行单元的地址。

我们已经解释了图2-18中除RETURN以外的所有语句。RETURN是一个汇编指示，它注明在这个点上执行程序应该结束。它使得汇编程序在这一点上插入一条适当的机器指令，该指令将控制权返还给计算机的操作系统。

许多汇编语言要求源程序语句的编写按以下方式进行：

Label    Operation    Operand(s)    Comment  
(标号) (操作)    (操作数)    (注释)

这四个字段采用适当的定界符，通常用一个或多个空格字符，进行分隔。标号Label是任选的一个与内存地址相关的名字，在此处机器语言指令从将要装入的语句中产生。标号也可以与数据

项的地址相关联。在图2-18中有五个标号：SUM、N、NUM1、START和LOOP。

操作字段中包含着所需指令或汇编程序命令的OP码助记符。操作数字段包含着用于访问的一个或多个操作数的地址信息，具体是一个或多个操作数取决于指令的类型。注释字段将被汇编程序所忽略，它是使程序更容易被理解的文档。

我们已经介绍的仅仅是汇编语言的最基本特性。任何一台计算机在这些语言细节上和复杂度上是不完全相同的。

## 2.6.2 程序的汇编和执行

用汇编语言编写的源程序在能够被执行之前必须被汇编成机器语言的目标程序。这是由汇编程序来完成的，它将用在机器指令中使用的二进制码替换所有表示操作数的符号以及寻址方式，并且用它们的实际值替换所有的名字和标号。

汇编程序为指令和数据块分配地址，起始地址由ORIGIN汇编指令给出。它还插入一些可能是在DATAWORD命令中给出，并由RESERVE命令要求保留在内存空间的常数。

汇编处理的一个关键部分是决定那些将要用来替代名字的值。在某种情况下，名字的值是由EQU指示说明的，这是一种简单的方式。在另一些情况下，名字是由一个给定指令的标号字段定义的，用这种名字表示的值是由汇编后的目标程序中这条指令所在的位置决定的。因此，汇编程序必须对连续指令在生成机器码时的地址保持了解。例如，名字START和LOOP将分别被分配的值是100和112。

在有些情况下，汇编程序不直接用这个地址的实际值去取代表示这个地址的名字。例如，在一个转移指令中，表示将要转入的那个转移位置（转移目标）的名字不是用实际地址替换的。转移指令通常在机器码中用相对地址方式指定转移目标的方法来实现，就像在2.5节中解释的那样。汇编程序计算转移偏移量，即到目标的距离，并将它放进机器指令中。

当汇编程序扫描一个源程序时，它保留所有名字以及它们在符号表中相对应的数据值。这样，当一个名字第二次出现时，就使用它在符号表中的值来替换它。当一个名字在给出它的值之前作为一个操作数出现时，便会引发一个问题。例如，如果需要一个向前转移操作时就发生了这种情况。汇编程序将不能确定转移目标，因为这个被引用的名字还没有记录在符号表中。解决这个问题的简单办法是让汇编程序对源程序做第二遍扫描。在第一遍扫描期间，它建立了一个完整的符号表。在这次扫描结束时，所有的名字都已经被分配了数值。然后汇编程序对源程序做第二遍扫描，并从符号表中对所有的名字做替换。这样的汇编程序叫做二遍扫描汇编程序。

汇编程序将目标程序存储在一个磁盘中。在它开始运行之前，目标程序必须装载到计算机的内存中。为了完成这项工作，另一个叫做装载程序的实用程序必须已经在内存中了。执行这个装载程序就是运行一系列的输入操作，需要将机器语言程序从磁盘中传递到内存的指定位置上。装载程序必须知道这个程序的长度和将要存放它的内存地址。汇编程序通常把这些信息放在目标程序的头部。对已经装载的目标码，装载程序就使用转移到可执行语句的第一条指令上的方法，使这个目的程序开始执行。这条指令的被调用地址已经包含在汇编语言程序中了，它是END汇编指令的操作数。汇编程序将这个地址保存在该磁盘上的目标程序的头部。

当目标程序开始执行后，它会一直进行到结束，除非在程序中有逻辑性错误。用户必须能够容易地发现错误，汇编程序能够监测并报告语法错误。为了帮助用户发现其他的程序设计错误，系统软件通常提供一个调试程序。这个程序使用户能够在一些感兴趣的点上停止目标程序的运行，去检查各种处理器寄存器和内存单元的内容。我们将在第4章中更详细地介绍程序的调试。

### 2.6.3 数的表示

在与数值打交道时,使用熟悉的十进制记数法通常是很方便的。当然,这些值是用二进制方式存储在计算机中的。有些情况下,直接指定二进制模式是比较方便的。大多数汇编程序允许数值按照不同的方式去表示,使用的约定由汇编语言的语法来定义。例如,考虑数字93,它用8位二进制数表示为01011101。如果这个值被当作立即操作数使用,它可以作为一个十进制数给出,就像如下的指令:

```
ADD #93, R1
```

或者用一个前缀符号(比如百分号)识别成二进制数,比如:

```
ADD # %01011101, R1
```

二进制数可以写成比较紧凑如十六进制方式的数,这时数的四位可以用一位十六进制数表示。十六进制表示法是BCD码的一种直接扩展,BCD码在附录E中给出。在BCD码中前十个模式0000、0001、0010、…、1001用数0、1、2、…、9表示,其余的六个4位模式1010、1011、…、1111用字母A、B、…、F表示。在十六进制表示法中,十进制数93变成5D。在汇编语言中,常常用一个美元符号做前缀表示一个十六进制数,这样前面这条指令应写成:

```
ADD # $5D, R1
```

## 2.7 基本输入/输出操作

在本章的前面章节中描述了机器指令和寻址方式,并假定那些在指令操作中使用的数据已经储存在内存中了。现在我们来查看数据在计算机的内存和外设之间传递的方法。输入/输出(I/O)操作很重要,并且它们的执行方法会对计算机的性能产生重要的影响。这一专题我们将在第4章中详细讨论。这里介绍一些基本的思想。

考虑一个这样的任务,从键盘读入一个输入字符然后在显示屏幕上产生一个输出字符。完成这个I/O任务的一种简单方法是使用众所周知的程序控制I/O方法。从键盘到计算机的实际传递速度是受用户打字速度所限制的,它不会超过每秒几个字符。而从计算机到显示器的输出传送速度是非常高的。这个速率根据字符传输中经过的计算机和显示设备之间的连接情况而决定,一般是每秒几千个字符。但是,它还是比处理器的速度慢很多,处理器每秒能执行几百万条指令。这种处理器和I/O之间速度的差异产生了在机制上要对它们之间的数据传送进行同步的要求。

解决这一问题的方法如下:在输出时,处理器发送第一个字符,然后就等待来自于显示器的表示已接收到这个字符的信号。然后再发送第二个字符,依次类推。来自键盘的输入使用类似的方法;处理器等待着一个标志信号,这个信号表示键盘上的一个字符键被敲击,并且它的编码已在一些与键盘相关联的缓冲寄存器中保存着,然后处理器去读这个代码。

如图2-19所示,键盘和显示器是独立的设备。在键盘上击打一个键的动作并不能自动地使相应的字符在显示屏上显示。在I/O程序中有一个指令块将字符传递到处理器中,而另一个相关的指令块完成该字符的显示。

考虑将一个字符代码从键盘传送到处理器中的问题。击打一个键,就可将相应的字符编码存进一个与键盘相关联的8位缓冲寄存器中。我们将这个寄存器称为DATAIN,如图2-19所示。为了通知处理器在DATAIN中有一个有效的字符,状态控制标志SIN被置为1。有一个程序监控SIN,当发现SIN被置为1时,处理器就去读DATAIN中的内容。当这个字符被传送到处理器时,SIN自

动地被清为0。如果在键盘中有第二个字符输入，SIN再次被置成1并且重复该处理过程。

当字符从处理器传送到显示器时，发生类似的处理过程。在这个传送过程中使用了缓冲寄存器DATAOUT和状态控制标志SOUT。当SOUT等于1时，表示显示器已准备好去接收一个字符了。在程序控制之下，处理器监控SOUT，并且当SOUT被设置为1时，处理器就将一个字符码传送到DATAOUT中。当一个字符被传送进DATAOUT后SOUT被清为0；当显示设备准备好接收第二个字符时，SOUT再次被置成1。缓冲寄存器DATAIN、DATAOUT以及状态标志SIN、SOUT都是通常我们所了解的设备接口电路中的一部分。用于各种设备的这些电路都需经过一条总线与处理器相连，如图2-19所示。

65

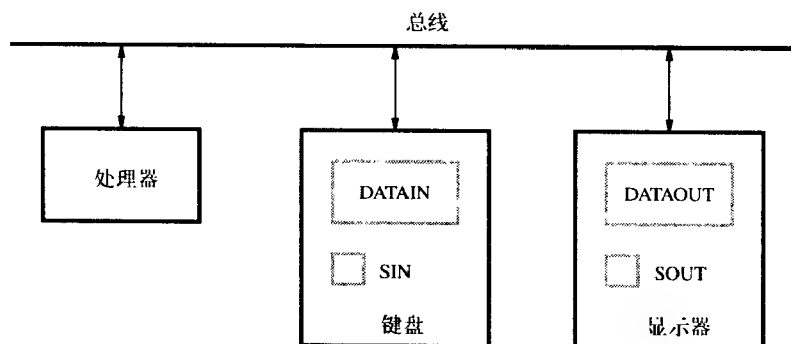


图2-19 处理器、键盘及显示器的总线连接

为了执行I/O传递，我们需要能够执行状态标志的检查和数据在处理器与I/O设备间传送的机器指令。这样的指令在形式上与那些在处理器与内存间传送数据的指令相似。比如，处理器能监控键盘状态标志SIN，并且用以下的操作序列将一个字符从DATAIN中传递到寄存器R1中：

READWAIT Branch to READWAIT if SIN = 0

Input from DATAIN to R1

转移操作通常用两条机器指令实现。第一条指令测试状态标志，第二条执行转移。虽然在计算机与计算机之间有一些细节上的不同，但主要的思想是处理器执行一个简短的循环等待实现对状态标志SIN的监控，并且当点击一个键而使SIN置为1时继续传递输入的数据。这个输入操作重新将SIN清0。

类似的操作序列可以用于将输出传送到显示器上，即：

WRITEWAIT Branch to WRITEWAIT if SOUT = 0

Output from R1 to DATAOUT

这里转移操作仍然用两条指令实现。循环等待重复执行直到状态标志SOUT被显示器置成1，这时表示显示器是空的，可以接收字符了。输出操作从R1向DATAOUT中传送一个字符用于显示，然后它将SOUT清为0。

我们假设SIN的初始状态是0，SOUT的初始状态是1。这个初始化工作通常是由设备控制电路执行的，它是在这些设备被置于计算机控制之下程序开始运行以前完成的。

到目前为止，我们假设处理器访问指令和操作数时，寻址过程总是访问内存单元。在许多计算机中使用了一个叫做存储器映射I/O的策略，在这个策略中一些内存的地址值被当作访问外围设备的缓冲寄存器，比如像DATAIN和DATAOUT等。这样，就不需要用特殊的指令去访问这些寄存器中的内容了；数据可以在这样的寄存器之间传递，而处理器使用的指令我们已经介绍

过了, 像Move、Load或Store等。例如, 用下面这条指令可以将键盘字符缓冲区DATAIN中的内容传递到处理器寄存器R1中:

MoveByte DATAIN, R1

类似地, 使用下面的指令可以将寄存器R1中的内容传递到DATAOUT中:

MoveByte R1, DATAOUT

当缓冲器DATAIN和DATAOUT被分别引用时, 状态标志SIN和SOUT被自动清除。MoveByte操作码表示这个操作数的大小是一个字节, 以便于将它与操作码Move做区分, 因为Move操作已经在字操作数的传递中被使用了。在图2-19中我们已经确定了两个数据缓冲器, 它们可以像两个内存单元一样被寻址。为了用同样的方式处理状态标志SIN和SOUT, 可以使用给它们分配不同地址的方式来实现。但是, 比较通用的方法是将SIN和SOUT保存在设备状态寄存器中, 每个设备对应一位。我们假定寄存器INSTATUS和OUTSTATUS的第三位 $b_3$ 分别对应于SIN和SOUT。刚才描述的读操作现在可以用以下机器指令序列来实现:

```
READWAIT Testbit    #3, INSTATUS
          Branch = 0 READWAIT
          MoveByte   DATAIN, R1
```

写操作可以用如下序列实现:

```
WRITEWAIT Testbit    #3, OUTSTATUS
          Branch = 0 WRITEWAIT
          MoveByte   R1, DATAOUT
```

Testbit指令测试在目标单元中某一位上的状态, 这里被测试位的位置是由第一个操作数指出的。如果这个被测试位等于零, 则转移指令的条件为真, 产生一个转向循环等待开始处的转移。当设备就绪时, 也就是说当这个测试位变成1时, 数据从输入缓冲器中读出或是被写入到输出缓冲器中。

图2-20给出的程序使用这两种操作从键盘上输入一行字符并将它们输出到一个显示设备中。当字符一个一个被读入时, 它们被储存在内存的一个数据区里, 然后回显到显示器上。当读到一个回车字符CR并且存储内容被发送到显示器中时, 这个程序结束。存储数据区第一个字节的

	Move	#LOC, R0	初始化指针寄存器R0, 使其指向内存中存放字符的第一个单元
READ	TestBit	#3, INSTATUS	等待一个字符输入到键盘缓冲器DATAIN中
	Branch=0	READ	将字符从DATAIN传送到内存中 (这将SIN清0)
	MoveByte	DATAIN, (R0)	
ECHO	TestBit	#3, OUTSTATUS	等待显示器准备就绪
	Branch=0	ECHO	
	MoveByte	(R0), DATAOUT	将刚才读入的字符移到显示器缓冲寄存器 (这将SOUT清0)
	Compare	#CR, (R0)+	检查刚才读入的字符是否为CR (回车符)。如果不是, 则转移回去读取另一个字符。同时, 递增指针来存储下一个字符
	Branch≠0	READ	

图2-20 读一行字符并将它显示出来的一段程序

地址，也就是存储该行的位置是LOC。寄存器R0用来指向这个存储区，并且它被程序中第一条指令用LOC地址进行了初始化设置。比较指令中的自动递增寻址方式在每读入一个字符并被显示后，对R0做递增处理。

程序控制I/O方式需要处理器连续地参与I/O的活动，在图2-20给出的程序中几乎所有的执行时间都用在了两个循环等待中了，直到处理器等待到了一个敲入的字符或是等待到了显示器变成可用状态。在这种情况下，应该尽量避免浪费处理器的执行时间。另一种基于中断的I/O技术使用可以用来改进处理器的利用率。这种技术将在第4章中讨论。

## 2.8 堆栈和队列

计算机程序中常常需要利用已有的子程序结构去执行一个特殊的子任务。为了构成这种控制结构以及形成主程序与子程序之间的信息连接，使用了一种叫做堆栈的数据结构。本节中我们将描述堆栈以及另一个与其紧密相关的称为队列的数据结构。

被程序操作的数据可以用多种方法来组织。我们已经接触了像表这样的数据结构，现在来考虑一种重要的称为堆栈的数据结构。堆栈是一个数据元素表，其中的元素通常是字或是字节，它具有的访问约束是只有在该表的一端数据元素可以被增加或是被移出。这一端叫做该栈的栈顶，另一端为栈底。这种结构有时也称为下推栈。设想自助餐厅里的一叠盘子，客户从这叠盘子的顶部拿走新盘子，而清洗干净的盘子又被添加到这叠盘子的顶部。另一种称为后进先出(LIFO)栈，也是用来描述这种类型的存储机制的；它表示最后放入堆栈中的数据项在检索开始时是第一个被取出的。术语压入和弹出分别用来描述向栈中放入一个新的数据项和从栈的栈顶取出一个数据项。

储存在计算机内存中的数据可以被组织成一个栈，其中连续的元素占据连续的存储单元。假设第一个元素放在BOTTOM位置上，当新的元素被压进栈时，它们被放置在紧挨着的低地址单元中。在我们的讨论中，使用了一个按照减小存储地址方向进行增长的栈，因为这是一种惯用的方式。

图2-21给出了一个在计算机内存中保存字数据项的堆栈。它所保存的数字值以43为栈底，-28为栈顶。有一个处理器寄存器用来保存栈中元素地址的轨迹，它在任何时候都指向栈顶。这个寄存器叫做栈指针(SP)。它可以是一个通用寄存器或是一个专门用于这一功能的专用寄存器。如果假设一个按字节寻址的存储器具有32位的字长，压栈操作可以这样实现：

Subtract #4, SP

Move NEWITEM, (SP)

这里Subtract指令从保存在SP中的目标操作数中减去源操作数4，再把结果放回SP中。第二条指令从单元NEWITEM中传送一个字到该栈的栈顶上，在传送之前对栈指针减4。弹出操作可以这样实现：

Move (SP), ITEM

Add #4, SP

这两条指令从栈中将栈顶值移出到单元ITEM，然后将栈指针加4，使其指向新的栈顶元素。图2-22给出了在图2-21中的栈中完成这些操作的效果。

如果处理器具有自动递增和自动递减寻址方式，压栈操作可以用单条指令执行：

Move NEWITEM, -(SP)

68

69



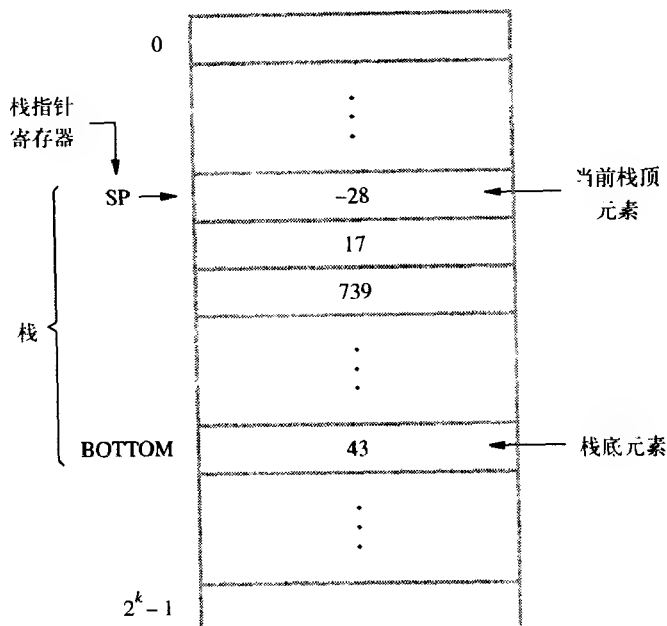
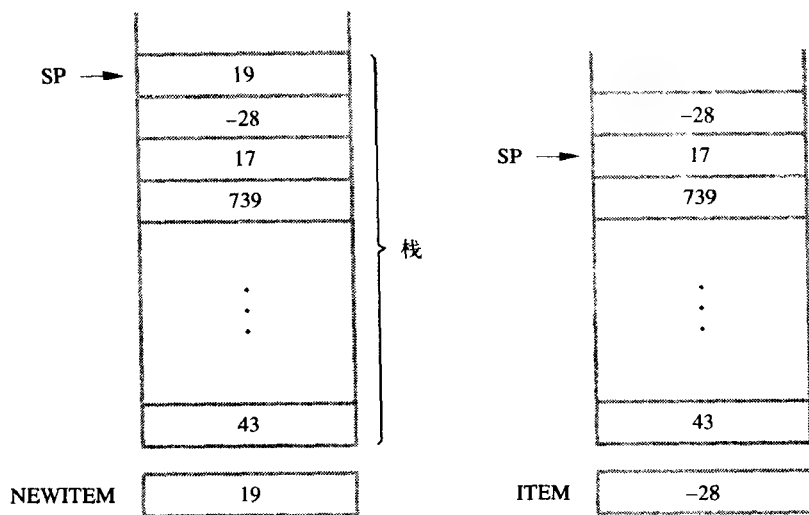


图2-21 存储器中一个字的堆栈



a) 将NEWITEM中的内容压入栈中

b) 弹出到ITEM后

图2-22 在图2-21的栈中栈操作的效果

弹出操作也能用下面一条指令来完成:

Move (SP)+, ITEM

当在一个程序中使用堆栈时,通常是在内存中划分出一个固定数量的存储空间。在这种情况下,当栈已经达到它的最大限定值时,必须避免再向栈中压入数据项的操作。同样,也要避免企图从空栈中弹出数据项的操作,这些操作将产生一个程序设计错误。假设有一个栈从单元2000 (BOTTOM) 向下运行,最大不能超过单元1500的位置。这个栈指针用地址值2004进行初

始化，在新数据被存进栈之前SP的内容要减4。因此，初始值2004表示第一个被压入栈中的数据项将是在单元2000处。为了防止向一个已满的栈上压入数据项或是从空栈中弹出一个数据项，单独的进栈和退栈操作指令可以用图2-23中给出的指令序列来代替。

比较指令

```
Compare src, dst
```

执行操作

```
[ dst ] - [ src ]
```

然后根据产生的结果设置条件码标志。但是这个操作不改变任何一个操作数的值。

70

SAFEPOP	Compare	#2000,SP	检查栈指针包含的地址值是否大于2000。如果是，栈为空。转移到EMPTYERROR例程采取适当的动作
	Branch > 0	EMPTYERROR	
	Move	(SP)+,ITEM	否则，将栈顶弹出到存储单元ITEM中

a) 用于安全弹出操作的程序

SAFEPUSH	Compare	#1500,SP	检查栈指针包含的地址值是否等于或小于1500。如果是，栈已满。转移到FULLERROR程序采取适当的动作
	Branch ≤ 0	FULLERROR	
	Move	NEWITEM, - (SP)	否则，将存储单元NEWITEM中的内容压入到栈中

b) 用于安全压入操作的程序

图2-23 在弹出和压入操作中检查空栈和栈溢出错误

另一个有效的类似于堆栈的数据结构叫做队列。数据被存入和从一个队列中取出是按照先进先出（FIFO）的原则进行的。这样一来，如果我们假设这个队列按照内存的增加地址方向进行增长，这是一种惯例，新的数据被加在队列的后面（高地址端），而检索是从队列的前面（低地址端）进行的。

71

在堆栈和队列的实现中有两个重要的不同之处。栈的一端是固定的（栈底），而另一端随着数据被压入和弹出而上升和下降，需要一个单指针指出任一给定时间的栈顶位置。而另一方面，当在后面增加数据并从前面移走数据时，队列的两端都要移向高地址，所以需要两个指针来保存两个端点的轨迹。

堆栈和队列之间的另外一个不同之处就是，队列可以连续地沿计算机存储器向高地址方向

移动,对这一点没有进一步的控制。将队列限制在内存一个固定范围内的方法是使用一个环形缓冲区。我们假定内存地址从BEGINNING到END被分配作为一个队列,该队列的第一项被输入到单元BEGINNING中,然后后续项用连续输入到高地址方向的方式被添加到队列中。当队列到达END返回时,如果有些项已经从队列中移出,在开始处就产生了一些空格。因此,返回指针复位到BEGINNING值上,并且继续处理过程。在堆栈情况下,一定要注意检测何时分配给数据结构的空间是完全满的或完全空的(请看习题2.18和2.19)。

## 2.9 子程序

在一个给出的程序里,常常需要使用不同的数据值多次执行一个特殊的子任务。这样的子任务通常叫做子程序。例如,一个子程序可以对正弦函数求值或者是将一个价值表按照递增或递减顺序分类排序。

可以包含有构成子程序的指令块,这些指令块可以在程序中任何需要的地方使用。可是为了节省空间,只有一个构成子程序的指令拷贝存放在内存中,任何需要使用该子程序的程序只是简单地转移到该子程序的起始位置即可。当一个程序将控制转移到一个子程序时,我们称其为调用子程序。执行这个转移操作的指令叫做调用(Call)指令。

一个子程序被执行后,调用它的程序必须继续执行,执行紧连在这条调用子程序指令后面的指令。在子程序中用执行一个Return指令的方式返回到调用它的程序中。因为子程序可能在一个调用程序的任何位置上被调用,所以必须准备好返回到适当的位置上。调用程序重新恢复执行的位置是当执行Call指令时被修改的PC所指出的位置。因此,为了使调用Call指令能够正确地返回到调用程序中,必须保存PC中的内容。

一个在计算机中能够调用并能够从子程序中返回的方法称为子程序链接法。最简单的子程序链接法是将返回地址存储在一个指定的单元中,这个单元也可以是一个专门用于这种功能的寄存器,这种寄存器称为链接寄存器。当子程序完成了它的任务时,Return指令用通过链接寄存器的间接转移返回到调用程序中。

Call指令只是一个特殊的转移指令,它执行以下的操作:

- 将PC中的内容存储到链接寄存器中。
- 转移到由这条指令指定的目标地址中。

Return(返回)指令是一条特殊的转移指令,它执行的操作是:

- 转移到链接寄存器中保存的地址中。

图2-24说明了这一进程。

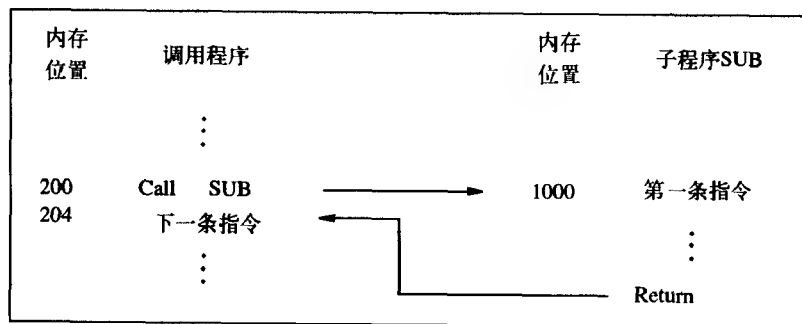


图2-24 使用链接寄存器的子程序链接

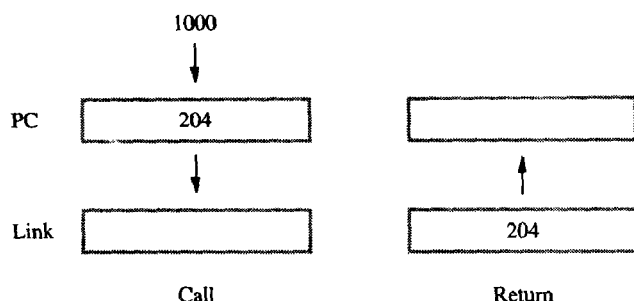


图2-24 (续)

### 2.9.1 子程序嵌套及处理器堆栈

使用一个子程序调用另一个子程序的程序叫做子程序嵌套。在这种情况下，第二个调用的返回地址也被存储在链接寄存器里，这就会破坏它原有的内容。因此，在调用其他子程序之前，把链接寄存器中的内容存储到其他的单元里是十分重要的，否则第一个子程序的返回地址将会丢失。

子程序的嵌套可以达到任何一种深度。最终，上一个被调用的子程序完成它的计算并返回到调用它的子程序中。这个前一次返回所需要的返回地址，是上一个调用按照嵌套调用顺序生成的。也就是说，返回地址的生成和使用是按照后进先出的顺序进行的。这使人联想到，与子程序调用有关的返回地址应该被压入到一个栈中。许多处理器将这个处理作为一个由Call指令执行的操作自动完成。被指定成栈指针的特殊寄存器SP将用在这个操作中。这个栈指针指向处理器栈。Call指令将PC中的内容压入处理器栈中，然后将子程序地址装入到PC中；Return指令从处理器栈中将返回地址弹出到PC中。

73

### 2.9.2 参数传递

当调用一个子程序时，程序必须要给子程序提供参数，也就是那些将在计算中使用的操作数或是它们的地址。然后，子程序返回另外的一些参数，在本例中是计算出的结果。这种在调用程序和子程序之间信息的交换称为参数传递。参数传递能够用多种方法实现。参数可以放在寄存器或存储器单元中，这些地方是能够被子程序访问到的。或者，这些参数可以放在用来保存返回地址的处理器栈中。

通过处理器寄存器做参数传递是简单有效的。图2-25给出了在图2-16的程序中如何将添加一个数字表的过程作成子程序，通过寄存器传递参数来实现。这个表的大小为 $n$ ，存储在内存单元N中，第一个数的地址NUM1，通过寄存器R1和R2被传递。这个和在子程序中计算出来并通过寄存器R0传回给调用程序。在图2-25中，前四条指令组成了调用程序的相关部分。其中的前两条指令把 $n$ 和NUM1装入到R1和R2中。Call指令转移到子程序的起始位置LISTADD处，这条指令还将返回地址压入到处理器栈中。子程序计算出这个和并将它放入到R0中。在返回操作被子程序执行后，这个和被调用程序储存到内存单元SUM中。

74

如果涉及到许多的参数，那么可用的通用寄存器就有可能不能满足向子程序传递参数的需要。而另一方面，栈的使用是非常灵活的，一个栈可以处理大量的参数。以下的例子说明了这种方法。图2-26a给出了将图2-16写成一个子程序LISTADD的程序，这个子程序可以被任意的程序调用去完成对一个数字表求和。传递给这个子程序的参数是该列表中的第一个数据地址以及

列表中的项数，子程序执行加法运算并返回计算的和，参数被压入到由寄存器SP指向的处理器栈中。假设在子程序被调用之前，栈顶在图2-26b中的第一级上。调用程序将地址NUM1和数值 $n$ 压入栈中，并调用子程序LISTADD。Call指令也把返回地址压入栈中，现在栈顶是在第二级上。

调用程序			
	Move	N,R1	R1作为计数器
	Move	#NUM1,R2	R2指向列表
	Call	LISTADD	调用子程序
	Move	R0,SUM	保存结果
	:		
子程序			
LISTADD	Clear	R0	将和初始为0
LOOP	Add	(R2)+,R0	累加列表项
	Decrement	R1	
	Branch>0	LOOP	
	Return		返回到调用程序

图2-25 图2-16中的程序写成一个子程序；通过寄存器传递参数

这个子程序使用了三个寄存器，因为这些寄存器中可能包含属于调用程序的有效数据，它们的内容应该被压入栈中进行保存。我们已经使用了单指令MoveMultiple将寄存器R0到R2的内容储存到这个堆栈中。许多处理器都有这样的指令，现在栈顶是在第三级上。子程序使用变址寻址方式从这个堆栈里访问参数 $n$ 和NUM1。注意这一操作不改变栈指针，因为有效的数据项仍然是在这个栈的栈顶。数值 $n$ 加载到R1里作为计数的初始值，并且地址NUM1装入到R2中，它当作扫描这个列表条目的一个指针。在计算结束时，寄存器R0中包含着计算的和。在子程序返回到调用程序之前，R0的内容被放入到这个栈中，替换了参数NUM1，因为这个参数不再需要了。然后这三个被子程序使用的寄存器内容从栈中被还原。此时栈顶中是第二级的返回地址。子程序返回后，调用程序将结果存在单元SUM中，再用SP增加8的方式将栈顶降到它原来的级别上。

#### 按值和按地址的参数传递

注意在图2-25和图2-26中两个参数NUM1和 $n$ 的实际值被传递到了子程序中，子程序的目的是为了对一个数字列表做加法。调用程序不传递实际的列表项，而传递在这个列表中第一个数的地址，这种技术叫做按地址传递。第二个参数是按值传递，也就是项数的实际数字 $n$ 被传递到子程序中。

### 2.9.3 堆栈的结构

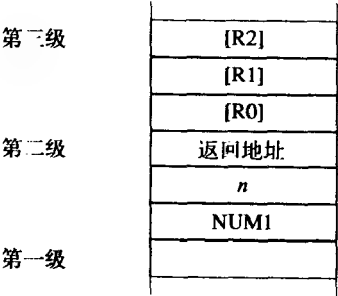
现在来看在图2-26的例子中存储空间是如何作为栈来使用的。在子程序执行期间，栈顶的6个单元包含了子程序需要的条目。这些单元构成了一个子程序的私有工作空间，当子程序进入时创建，当子程序将控制返回到调用程序时被释放。这种空间叫做栈结构。如果子程序需要更多的空间用于局部存储变量，它们也可以用栈进行分配。

图2-27给出了一个对于栈结构中的信息经常使用的布局例子。除了堆栈指示器SP外，通常还需要另一个指针寄存器，叫做结构指针（FP），它是为了方便对传递到子程序中的参数进行访问，以及便于对子程序中使用的局部变量进行访问的。这些局部的变量只能在该子程序中使用，因此对于它们适合使用与子程序相关的栈结构进行存储空间的分配。在该图中，我们假设有四个

参数被传递到子程序中，有三个局部变量要在子程序内部使用，还有寄存器R0和R1需要进行存储，因为它们也要在子程序中被使用。

假设栈顶在第一级			
	Move	#NUM1, -(SP)	保存参数到栈中
	Move	N, -(SP)	
	Call	LISTADD	调用子程序
			(栈顶在第二级)
	Move	4(SP), SUM	保存结果
	Add	#8, SP	恢复栈顶
			(栈顶在第一级)
	:		
LISTADD	MoveMultiple	R0-R2, -(SP)	保存寄存器
			(栈顶在第二级)
	Move	16(SP), R1	将计数器初始化为n
	Move	20(SP), R2	初始化指针使其指向列表
	Clear	R0	将和初始化为0
LOOP	Add	(R2)+, R0	累加列表条目
	Decrement	R1	
	Branch > 0	LOOP	
	Move	R0, 20(SP)	将结果保存到栈中
	MoveMultiple	(SP)+, R0-R2	恢复寄存器
	Return		返回到调用程序

a) 调用程序和子程序



b) 不同时刻的栈顶

图2-26 将图2-16中的程序写成一个子程序；在堆栈上传递参数

如图2-27中给出的那样，使用FP寄存器指出的单元正是上述的存储返回地址，我们可以使用变址寻址方式很容易地访问到这些参数和局部变量。这些参数可以用地址8 (FP)、12 (FP)、...访问到。局部变量可以用-4 (FP)、-8 (FP)、...访问到，FP中可以保留子程序运行中的任何一个单元，这一点不同于栈指针SP必须总是指向栈中当前的栈顶元素。

现在来讨论指针SP和FP，当栈结构为了该子程序的一个特殊请求而被建立、被使用和被拆除时是如何操作的。在图2-27中我们开始假定SP指向原来的栈顶单元，在子程序被调用之前，调用程序将四个参数压入栈中。然后当执行Call指令时，结果是返回地址被压入栈中。现在SP指向了这个返回地址，并且子程序的第一条指令将要被执行。这是指向被设置成包含有适当内存地址的结构指针FP。因为FP通常是一个通用寄存器，它可能包含有用于调用程序的信息。因此，它的内容被压入栈中进行保存。因为SP现在指向这个位置，它的内容被拷贝到FP中。

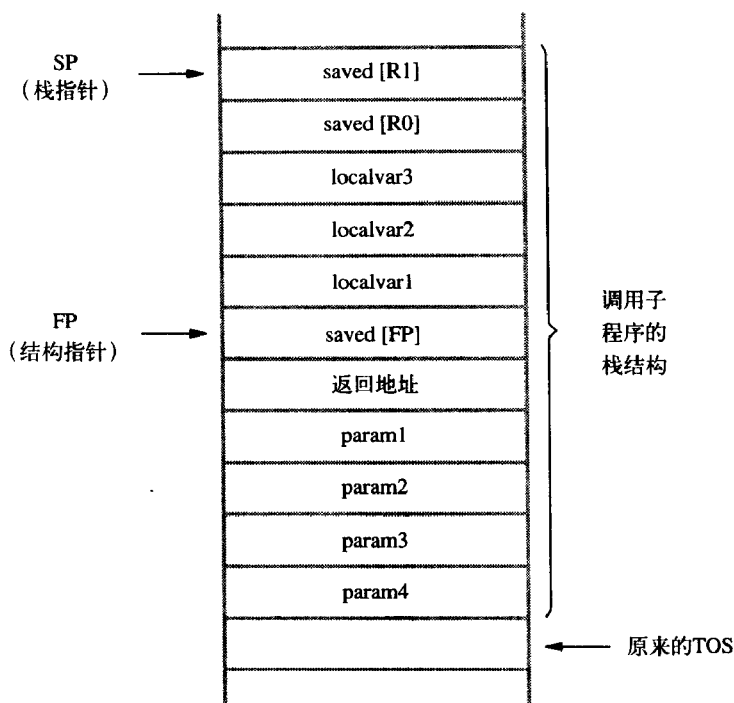


图2-27 一个子程序栈结构的例子

这样一来，在子程序中前两条指令执行的是：

```
Move FP, -( SP )
```

```
Move SP, FP
```

这些指令执行以后，SP和FP都指向了保存后的FP的内容。现在三个局部变量的空间被以下指令的执行分配到了栈中：

```
Subtract #12, SP
```

最后，处理器寄存器R0和R1的内容被压入栈中进行保存。此时，这个栈结构已经被设置成图中所示的状况。

子程序现在开始执行它的任务。当这个任务完成时，子程序弹出R1和R0的保存值放回到这些寄存器中，执行以下的指令从栈结构中删除局部变量：

```
Add #12, SP
```

并且弹出保存的FP旧值放回到FP中。这时候，SP指向了返回地址，所以返回指令Return可以执行了，将控制返还给了调用程序。

调用程序负责从栈结构中删除这些参数，其中一些可能是子程序返回来的结果。栈指针现在指向了原来的TOS（栈顶），又返回到了起始点。

#### 用于子程序嵌套的栈结构

当子程序嵌套时，堆栈是用于处理返回地址的最合适的数据结构。应该很清楚，当子程序被调用时为了嵌套子程序在处理器栈上建立了完整的栈结构。在这一点，应注意在当前结构中栈顶中存储的FP的内容是结构指针的内容，这个结构指针是用于调用当前子程序的子程序栈结构。

我们来看一个例子，主程序调用了第一个子程序SUB1，然后在其中又调用了第二个子程序

SUB2, 该过程由图2-28给出。对应于这样的两个嵌套子程序的栈结构显示在图2-29中。所有与这个例子相关的参数都被传递到栈中, 该图中只显示了在这三个程序中的控制流和数据。实际的计算方法没有给出。

执行的流程如下: 主程序将两个参数param2和param1按顺序压进栈中, 然后调用SUB1。这里第一个子程序负责计算一个答案并使用栈将这个答案传回主程序。在它的计算过程中, SUB1调用了第二个子程序SUB2, 用于执行一些子任务。SUB1给SUB2传递了一个单独的参数param3, 并且获得一个给它传回来的结果。当SUB2执行了它的Return指令以后, 这个结果被SUB1存储到了寄存器R2中。SUB1继续它的计算, 最后用堆栈方式将所需要的答案传回给主程序。当SUB1执行返回指令返回到主程序时, 主程序将这些答案存储在内存单元RESULT中, 并且使用它继续计算的步骤在“下一条指令”中。

内存位置		指令	注释
<b>主程序</b>			
		⋮	
2000		Move PARAM2, -(SP)	将参数放置到栈中
2004		Move PARAM1, -(SP)	
2008		Call SUB1	
2012		Move (SP), RESULT	保存结果
2016		Add #8, SP	恢复栈级
2020		next instruction	
		⋮	
<b>第一个子程序</b>			
2100	SUB1	Move FP, -(SP)	保存结构指针寄存器
2104		Move SP, FP	载入结构指针
2108		MoveMultiple R0-R3, -(SP)	保存寄存器
2112		Move 8(FP), R0	获取第一个参数
		Move 12(FP), R1	获取第二个参数
		⋮	
		Move PARAM3, -(SP)	将一个参数放置到栈中
2160		Call SUB2	
2164		Move (SP)+, R2	弹出SUB2的结果到R2中
		⋮	
		Move R3, 8(FP)	将答案放入栈中
		MoveMultiple (SP)+, R0-R3	恢复寄存器
		Move (SP)+, FP	恢复结构指针寄存器
		Return	返回主程序
<b>第二个子程序</b>			
3000	SUB2	Move FP, -(SP)	保存指针寄存器
		Move SP, FP	载入结构指针
		MoveMultiple R0-R1, -(SP)	保存寄存器R0和R1
		Move 8(FP), R0	获取参数
		⋮	
		Move R1, 8(FP)	将SUB2的结果放入栈中
		MoveMultiple (SP)+, R0-R1	恢复寄存器R0和R1
		Move (SP)+, FP	恢复结构指针寄存器
		Return	返回到第一个子程序

图2-28 嵌套子程序



在图2-28注释中给出了如何管理这个流程的细节内容。每个子程序执行的第一个动作是设置结构指针，保存了前一个栈中内容以后，再保存其他所需要寄存器的内容。SUB1使用了R0到R3的四个寄存器，而SUB2使用了两个寄存器R0和R1。这些寄存器和结构指针只有在执行返回指令之前被恢复。

使用与结构指针寄存器FP有关的变址寻址方式，可以从栈中加载参数以及将答案放回到栈中。在操作中使用的字节偏移量，通常是8、12、16、...，如同在图2-27中讨论的一般栈结构一样。最后要注意的是，调用程序要负责从栈中删除参数。这是由主程序中的Add指令和在SUB1单元2164中的Move指令来完成的。

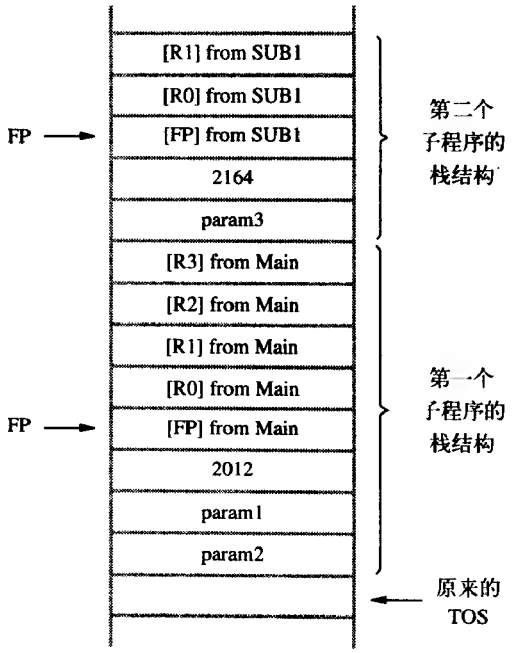


图2-29 图2-28的栈结构

## 2.10 附加的指令

到目前为止，我们已介绍了下列的指令: Move、Load、Store、Clear、Add、Subtract、Increment、Decrement、Branch、Testbit、Compare、Call和Return。使用这13条指令和表2-1中的寻址模式，已经可以编写程序例子来说明机器指令的执行序列了，包括转移和子程序结构。我们还说明了基本的存储器映射I/O操作。

这个小指令集有许多的冗余，比如Load和Store指令能够用Move指令代替，Increment和Decrement指令能够分别用Add和Subtract指令代替，而Clear指令可以用包含一个立即操作数0的Move指令代替；因此，只用余下的8条指令就已经可以满足我们的目的了。但是，在实际的机器指令系统中存在一些冗余是很正常的，一些简单的操作通常能够用多种不同的方法来实现。有一些可选择的方法可能会比其他方法更有效。在这一节中，我们介绍一小部分比较重要的指令，这些指令是在大部分指令集中都存在的。

### 2.10.1 逻辑指令

像AND（与）、OR（或）和NOT（非）这样的逻辑操作，适合于对那些数字电路基本构件的按位操作，就像在附录A中介绍的那样。它也可以用于软件执行中的逻辑操作，这些操作可以用指令来完成，这些指令可以将这些操作独立或是并行地施加于一个字或一个字的所有位上。例如以下指令：

Not dst

是对保存在目的操作数中的所有位求反，即将1变成0，0变成1。在2.1.1节中，我们看到将1加到有符号正数的反码格式上就形成了一个补码格式表示的负数值。例如在图2-1中，通过给0011的反码格式加了1，使+3（0011）转换成了-3（1101）。如果3存储在寄存器R0中，那么指令：

Not R0  
Add #1, R0

就完成这个转换。很多计算机中有一个单指令：

Negate R0

可以完成同样的事情。

现在考虑逻辑指令AND（与）的应用，它在源操作数和目的操作数上执行按位AND（与）操作。假设有四个ASCII字符被保存在32位的寄存器R0中。在某个任务中，我们希望确定最左边的字符是否为Z，如果是Z，就执行条件转移到YES。从附录E中我们找到了字符Z的ASCII码是01011010，用十六进制记数法表达为5A。这三条指令的序列

81

```
And      #$FF000000, R0
Compare  #$5A000000, R0
Branch=0 YES
```

实现了所期望的动作。And指令将R0中右边三个字符清0，而使最左边的字符不变。这是使用一个立即源操作数的结果，该操作数的左端有8位是1，后面的24位是0。比较指令Compare用字符Z的二进制表示形式同R0中左端的剩余字符进行比较。如果有相匹配的值，Branch指令产生一个转向YES的转移指令。

And指令经常用在当一个操作数中除了一些特殊区域，所有的位都要清为0的实际程序设计任务中。在我们的例子里，R0中最左边8位构成了特殊的区域。

## 2.10.2 移位和循环移位指令

有很多应用需要将一个操作数的位向左或向右移动一些指定数量的位。这个移动过程如何被执行，取决于操作数是一个有符号数或者是一般普通二进制码信息。对于一般的操作数，我们使用逻辑移位。对于一个数字，我们使用算术移位，算术移位可以保护该数字的符号不变。

### 逻辑移位

需要两种逻辑移位指令，一个是左移（LShiftL），另一个是右移（LShiftR）。这些指令将一个操作数移动一定数量的位，这个移动数量值由在该指令中的一个计数操作数说明。一个逻辑左移指令的一般形式为：

LShiftL count, dst

计数操作数可以作为一个立即操作数给出，或者可以作为处理器寄存器的内容给出。为了完善左移操作的描述，我们需要指明目的操作数右边空出位的值，并确定在左端移出的位上发生了什么。空出的位置用0填充，并且移出的位经过进位标志C，然后再丢弃。将C标志一起参与移位，这在执行算术操作时一个大数占用了比一个字还要多的位置时是有用的。图2-30a给出了一个例子，它将寄存器R0的内容左移两位。逻辑右移指令LShiftR除了向右移以外，其他工作方式是相同的。图2-30b说明了这种操作。

### 数字打包举例

考虑以下的小任务，它说明了移位操作和逻辑操作的用法。假设两个ASCII码表示的十进制数存储在内存单元字节LOC和LOC+1中。我们希望将这些十进制数用4位BCD码表示，并将它们存放到一个单字节单元PACKED中。上述的结果是要形成打包BCD码格式。附录E中的表E-1和E-2给出了十进制数ASCII码相应于这个数的BCD码最右边的四位数。因此，所需的任务是在LOC和LOC+1中提取出低4位的内容，然后将它们拼接成一个单字节放在PACKED中。

82

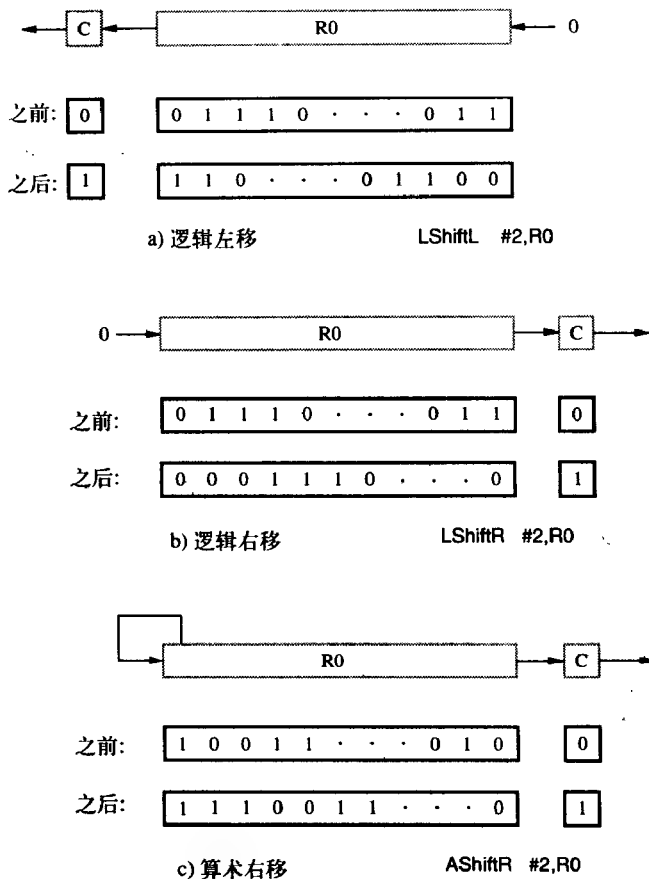


图2-30 逻辑和算术移位指令

图2-31给出的指令序列完成这个任务，它用寄存器R0作为指向内存中ASCII码字符的指针，并用寄存器R1和R2生成BCD数字码。当一个MoveByte指令在存储器和32位处理器寄存器之间传递一个字节时，我们将这个字节安排到寄存器的最右边8位上。And指令用于屏蔽R2中最右边的四位。注意，立即源操作数被写成\$F，将它作为一个32位的模式解释，其主要的有效位上有28个0。

83

Move	#LOC,R0	R0指向数据
MoveByte	(R0)+,R1	将第一个字节载入R1
LShiftL	#4,R1	左移4位
MoveByte	(R0),R2	将第二个字节载入R2
And	\$F,R2	清除高位位
Or	R1,R2	连接BCD数
MoveByte	R2,PACKED	保存结果

图2-31 一个对两个BCD数打包的程序

### 算术移位

图2-1中补码的二进制数表示法表明将一个数向左移动一位就等于这个数乘以2；而向右移动一位等于这个数除以2。当然，在左移时可能会产生溢出，而右移时余数可能会被丢失。另一个要注意的是在右移中符号位在对空位进行填充时必须重写。这需要在右移处理中将算术移位与逻辑移位加以区别，在逻辑移位中总是用0做空位填充。没有这些的话，这两种移位是非常相

似的。在图2-30c中给出了一个算术右移AShiftR的例子，算术左移与逻辑左移是完全相同的。

### 循环移位操作

在移位操作中，除了最后一个移出位保存在进位标识C中以外，操作数的移出位都被丢掉了。为了保留所有的位，可以使用一套循环指令。它们将操作数一端移出的位返回到操作数的另一端。通常提供循环左移和循环右移两种版本。在一种版本里，操作数的位被简单地进行循环。在另一种版本里，循环中包括了C标志。图2-32给出了C标志包含或不包含在循环中的循环左移和循环右移操作。要注意的是，当循环中不包括C标志时，它仍然保留着寄存器尾部的最后一个移出位。助记符RotateL、RotateLC、RotateR和RotateRC表示执行循环移位操作的指令。Rotate指令主要用于执行算术运算的算法过程中，而不是完成加法和减法，有关这点我们将在第6章中再次谈到。

84

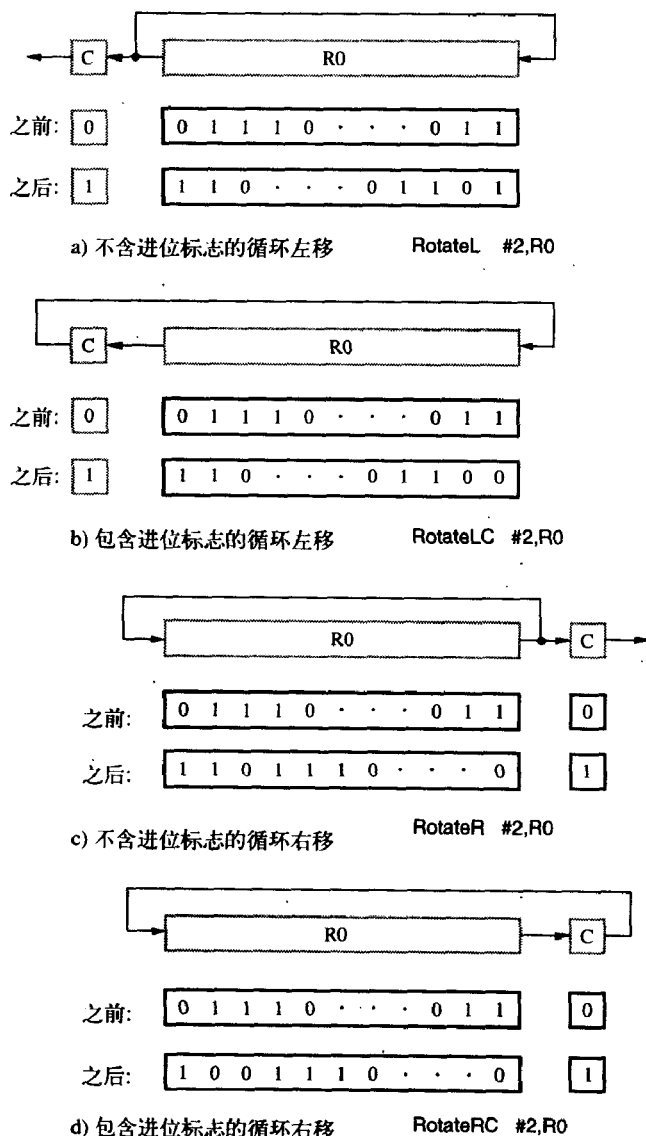


图2-32 循环移位指令

85

### 2.10.3 乘法和除法

两个有符号整数能够使用机器指令像用Add指令一样的方式进行乘或除。指令

Multiply Ri, Rj

执行的操作为:

$$Rj \leftarrow [Ri] \times [Rj]$$

两个 $n$ 位数的乘积可以是 $2n$ 位大的数。因此,答案肯定不能放进 $Rj$ 中。大多数指令系统中有一个乘法指令,它计算乘积的低 $n$ 位并把它放在寄存器 $Rj$ 中,就像说明的那样。如果已知在一些特殊的应用任务中所有乘积都将是不超过 $n$ 位的数,这是足够的。为了适应一般的 $2n$ 位乘积的情况,一些处理器将乘积生成在两个寄存器中,通常是相邻的寄存器 $Rj$ 和 $R(j+1)$ ,其中高位部分放在寄存器 $R(j+1)$ 中。

虽然不是所有的,但有些指令系统还提供了有一个有符号整数的除法(Divide)指令

Divide Ri, Rj

它执行的操作是:

$$Rj \leftarrow [Rj] / [Ri]$$

把得到的商放到 $Rj$ 中。余数可以放在 $R(j+1)$ 中,或者可以丢掉。

对于没有乘法和除法指令的计算机,可以用一些基本的指令序列比如加(Add)、减(Subtract)、移位(Shift)和循环移位(Rotate)来完成这些以及其他的算术操作。当我们在第6章中描述算术操作的实现时,这些将会变得更清楚。

## 2.11 实例程序

在这一节里,我们给出三个例子来进一步说明机器指令的使用。这些例子代表了数字的(向量处理)和非数字的(排序和链表操作)的应用。

### 2.11.1 向量点积程序

第一个例子是有关于数字的应用,它是图2-16用于加数的循环程序的一个扩展格式。在计算中包括了向量和矩阵,它对于计算两个向量的点积通常是必要的。假设A和B是两个长度为 $n$ 的向量。它们的点积被定义为:

$$\text{Dot Product} = \sum_{i=0}^{n-1} A(i) \times B(i)$$

图2-33给出了一个计算点积并把它储存到内存单元DOTPROD中的程序。每个向量的第一个元素 $A(0)$ 和 $B(0)$ 被储存到内存单元AVEC和BVEC中,其余的元素就储存在随后的字单元中。

86

这种乘积累加和的任务常出现在信号处理的许多应用中。在这种情况下,其中的一个向量由最近的 $n$ 个信号采样组成,这些信号采样是按照输入到信号处理单元的连续时序完成的。另一个向量是 $n$ 个权重的集合。 $n$ 个信号采样值乘以这个权重,而这些乘积的总和组成了一个输出信号实例。

有些计算机指令系统将图2-33程序中的乘法和加法指令操作结合起来,成为一个单独的MultiplyAccumulate指令。在第3章的ARM处理器中将会看到这样的一个例子。

	Move	#AVEC,R1	R1指向向量A
	Move	#BVEC,R2	R2指向向量B
	Move	N,R3	R3作为计数器
	Clear	R0	R0累加点积
LOOP	Move	(R1)+,R4	计算下组元素的积
	Multiply	(R2)+,R4	累加到前面的和中
	Add	R4,R0	减小计数器
	Decrement	R3	如果没有完成再次循环
	Branch> 0	LOOP	将点积保存到内存中
	Move	R0,DOTPROD	

图2-33 计算两个向量点积的程序

### 2.11.2 字节排序程序

考虑将一个存储在内存中的字节列表分拣成按字母升序排序的方式。假设这个列表由 $n$ 个字节构成，但这不是严格的，并且在每个字节包含字母表A到Z字符的ASCII码。参见附录E，在ASCII码中字母A, B, ..., Z用7位模式表示，当这种模式作为二进制数解释时其值是逐渐递增的。当一个ASCII码字符被储存在一个字节单元时，按照惯例把最高有效位的位置设置成0。使用这种编码，我们可以采用将它们的编码按递增数字顺序排序的方法来分拣一个按字母排序的字符表，将字符编码都看成正数。

将这张表保存在内存单元LIST到LIST +  $n - 1$ 中，并且将 $n$ 作为一个32位的值储存在地址N中。分拣工作在原来位置上完成，也就是说，分拣后的表与该原表占据着同一部分的内存单元。

我们用一个直接选择排序算法对该表进行分类。首先，找出最大的数值并将它放到表的尾部位置LIST +  $n - 1$ 中。然后再把在剩下的 $n - 1$ 个数字子表中的最大数放在子表的末端位置LIST +  $n - 2$ 中。这个处理过程重复直到这个表被分拣完成。使用这种分拣算法的一个C语言程序在图2-34a中给出，在该程序中这个列表被当作一个从LIST(0)到LIST( $n - 1$ )的一维数组。对于每个LIST( $j$ )到LIST(0)的子表来讲，LIST( $j$ )里的数与该子表中的其他数进行比较。每当在子表中找到一个较大的数时，它就与表LIST( $j$ )中的数进行交换。

87

这个C语言程序向后遍历该列表。这种遍历调整方式简化了机器语言版本的程序中循环结束过程，因为当变址值被减为0时循环退出并结束的。

在图2-34b中给出了一个实现这种排序算法的汇编语言程序。程序中的注释解释了各种寄存器的用法。当前的最大值保存在寄存器R3中直到有一个子表被扫描。如果发现一个更大的值，就用它与R3中的值做交换并把新的最大值储存在LIST( $j$ )中。

88

在这两个程序中为了提高效率，使用汇编语言程序时其控制流的处理方法是不同的。在C语言程序中使用if-then控制语句，如果LIST( $k$ ) > LIST( $j$ )时，使得第三行的then子句完成LIST( $k$ )和LIST( $j$ )的交换。在汇编语言程序中，转移发生在如果LIST( $k$ ) < LIST( $j$ )成立时的四条交换代码指令附近。

如果本机器指令系统允许传送操作可以从一个存储单元直接到另一个存储单元时，则在图2-34b中内部循环的四条交换代码指令可以用三条指令序列取代：

```
MoveByte (R0, R2), (R0, R1)
MoveByte R3, (R0, R2)
MoveByte (R0, R1), R3
```

就像我们将在第3章中看到的那样, 68000处理器就有这样的能力。

最后注意到图2-34b中的程序, 只有当列表中至少有两个元素时才能正确地运行, 因为对循环终止的检测是在每个循环的最后进行的。因此, 不管 $n$ 的值是多少都至少通过了一遍的循环。

```

for (j = n-1; j > 0; j = j - 1)
{ for (k = j-1; k >= 0; k = k - 1)
  { if (LIST[k] > LIST[j])
    { TEMP = LIST[k];
      LIST[k] = LIST[j];
      LIST[j] = TEMP;
    }
  }
}

```

a) C语言排序程序

OUTER	Move	#LIST,R0	将LIST载入R0
	Move	N,R1	初始化外部循环变址
	Subtract	#1,R1	将寄存器R1值设为 $j = n - 1$
	Move	R1,R2	初始化内部循环变址
	Subtract	#1,R1	将寄存器R2的值设为 $k = j - 1$
INNER	MoveByte	(R0,R1),R3	将LIST( $j$ )载入R3, R3保存当前子列表的最大值
	CompareByte	R3,(R0,R2)	如果LIST( $k$ ) < [R3]不交换
	Branch<0	NEXT	
	MoveByte	(R0,R2),R4	否则, 交换LIST( $k$ )和LIST( $j$ )
	MoveByte	R3,(R0,R2)	并将新最大值载入R3
NEXT	MoveByte	R4,(R0,R1)	寄存器R4作为TEMP
	MoveByte	R4,R3	
	Decrement	R2	减小变址寄存器R2和R1, 它们
	Branch>0	INNER	也作为循环计数器, 当循环尚未
	Decrement	R1	结束时转移回去
	Branch>0	OUTER	

b) 汇编语言排序程序

图2-34 使用直接选择算法的字节排序程序

### 2.11.3 链表

许多非数字的应用程序需要一个存储在内存中表示信息项的顺序表, 在这种方式中很可能要完成在表的任意位置上添加项或是从表中删除项, 而仍然能维护所需要项的顺序。这与2.8节中讨论的堆栈和队列数据结构相比是一种更一般的情况, 在堆栈和队列的数据结构中只能在数据结构的两端进行添加和删除。考虑以下的例子, 在2.5节中用来说明变址寻址模式的学生课程成绩表中包含一个具有惟一性的学生ID号, 这个ID号是每个学生记录中四个字中的第一个字, 如图2-14所示。假设我们试图以学生ID号递增顺序、按相邻的内存块、以连续的存储单元来维护这张记录表。这样做可以使利用ID号来打印和邮寄考试成绩记录表的工作变得容易一些。在该表被建成以后, 如果有某个学生从课程中退出, 就会产生一条空记录的位置。当需要对所有记录做考试分数的累加或是需要打印出一张表时, 就需要跳过这条空记录。在这张表的初始结构建立后如果又有一个学生注册了这门课程, 就会出现一种比较尴尬的情况。为了保持该列表的顺序, 所有的记录中从第一个比新的ID号大的记录开始, 需要移到高一些的地址单元中去, 为新的记录创建一个四字长的空间。类似地, 当对以前有过的学生信息做撤销处理时, 由于空位

置的原因也要做移动,而这个移动要对这个空位以后的所有记录完成向低地址单元移动的操作才能弥补这个空缺。

链表的数据结构可以用来避免这两种问题。在链表中每条记录仍然占有内存中一个连续的四字长的块,但在顺序上连续的记录没有必要占用连续的内存地址空间块。为了将这些块连接在一起形成一个顺序表,每条记录中包含一个一字长的链接字段的地址值,它具体指出该顺序中下一条记录的具体位置,因此链表可以用来描述这种数据结构。关于一个链表的图表说明在图2-35a中给出。这个表中的第一条记录叫做表头,最后一条记录叫做表尾。

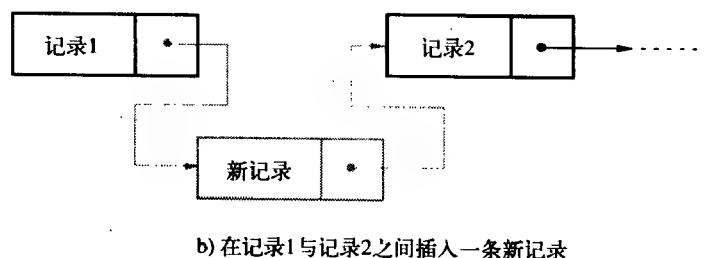
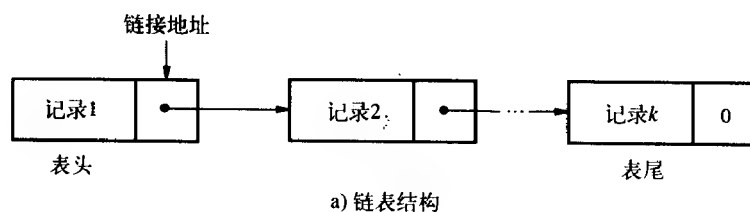


图2-35 链表数据结构

为了在记录 $i$ 和 $i+1$ 之间插入一条新记录,记录 $i$ 中的链接地址拷贝到新记录的链接字段中,然后新记录的地址写到记录 $i$ 的链接字段中。这个操作在图2-35b中给出了图示。为了删除记录 $i$ , $i$ 的链接字段地址被拷贝到记录 $i-1$ 的链接字段中。

图2-36给出了学生考试成绩记录在存储器中链接起来的例子,它的顺序是按递增的ID号排列的。每条记录现在是五个字长,第一个字作为关键字段定义,其中包含着学生的ID号。第二个字段包含的内容是链接字段,最后三个字段是数据字段,其中包含着三个考试成绩。假定字长是32位,起始字地址在1000处的存储器中的2000个字节空间,被分配用来保存每个学生的五字长记录,每条记录有20个字节,最多有100个学生。当学生注册课程时,在内存中分配一个可获得

90

的五字长记录块。按照块地址顺序1000, 1020, 1040, ..., 2980进行分配可能是方便的,但并不是必需的。在学生的ID号之间和学生对于课程注册顺序之间没有特殊的关联关系。因此记录块的位置、学生ID号的顺序将以一种不可预测的方式在块地址1000到块地址2980之间指定的存储空间中散布着。

具有当前最低ID号的记录居于该表的表头,而具有当前最高ID号的记录是在表的尾部。访问这个表的一种方便方法是将表头的存储地址(这里是2320),保存到处理器的头指针寄存器中。第一条记录的链接字段中的地址1040用于指明第二条记录的位置。第二条记录的链接字段包含着第三条记录的地址1200等。最后一条记录的链接字段设置成0,说明这是该表的表尾记录项。如果这个表是空的,头指针中的内容为0。

91



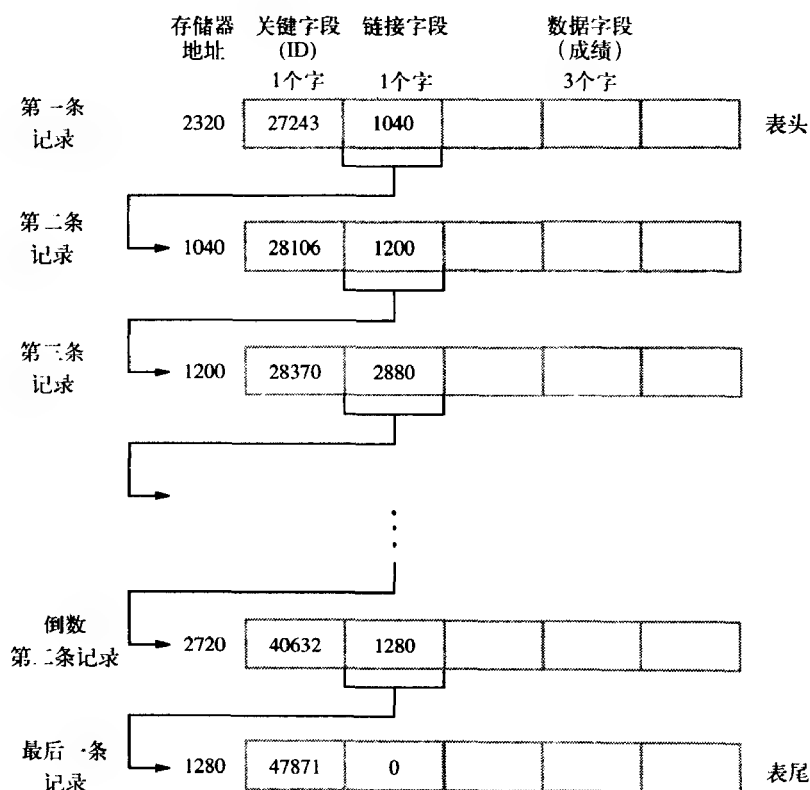


图2-36 学生考试成绩表作为一个链表在存储器中的结构

### 插入一条新记录

现在给出当需要将一条新记录添加到图2-36所示的表中的步骤。假定这条新记录的ID号是28241，而下一条可获得的空闲记录块在地址2960处，从头记录开始向前追踪直到找到一条具有更大ID号的记录为止。这里是存储单元1200中的记录，它包含的ID号是28370。现在将链接地址1200插入到新记录的链接字段中，然后将新记录的地址2960插入到单元1040中的前一条记录的链接字段中，覆盖原有的值1200。现在新记录已经被插入到了修改表的第三条记录中，在原来表的第二条和第三条记录之间。

完成插入操作的子程序在图2-37中给出。它由三部分构成，用来处理以下三种可能情况：当前表是空的，或者这条新记录变成了非空表的新表头，或者新记录插入到该表的当前头记录后面的某处位置上。最后一种情况包含着新记录变成表尾的情况。

现在考虑该子程序如何处理这三种可能情况。该子程序使用了多个处理器寄存器。为了替换常用名字R0、R1、R2等，我们用更具有描述性的名称来帮助理解。RHEAD是头指针，RNEWREC包含着新记录的地址。两个寄存器RCURRENT和RNEXT包含着当扫描该表找到用于插入新记录的正确位置的当前记录和下一条记录的地址。新记录的链接字段最初设置成0。如果它变成了新的表尾，这个字段不需要进一步改变。

第一个比较/转移指令检测该表是否为空，如果它是个空表（RHEAD的内容为0），将该记录的地址移到RHEAD中，使新的记录变成了一个表项的列表，后面紧跟着一个Return指令；否则，第二个比较/转移指令检查是否新的记录变成了表头。如果是，两个Move指令完成所需要的将新

记录链接字段和RHEAD中内容进行改变的工作，然后执行Return指令。如果新记录不是新的表头，子程序的后半部分确定在该表中何处是新记录将要被插入的位置。然后用最后的两个Move指令将它插入到正确的内部位置中，或是用最后的一个Move指令生成新的表尾。为了改善易读性和对插入操作理解，我们省略了在这个子程序中保存和恢复寄存器的过程。

不空	INSERTION	Compare	#0, RHEAD	
		Branch>0	HEAD	
		Move	RNEWREC, RHEAD	新记录成为一个表项的列表
		Return		
在当前表头后的某处插入新记录	HEAD	Compare	(RHEAD), (RNEWREC)	
		Branch>0	SEARCH	
		Move	RHEAD, 4(RNEWREC)	新记录成为新表头
		Move	RNEWREC, RHEAD	
		Return		
新记录成为新表尾	SEARCH	Move	RHEAD, RCURRENT	
	LOOP	Move	4(RCURRENT), RNEXT	
		Compare	#0, RNEXT	
		Branch=0	TAIL	
		Compare	(RNEXT), (RNEWREC)	
在链表内部插入一条新记录		Branch<0	INSERT	
		Move	RNEXT, RCURRENT	
		Branch	LOOP	
	INSERT	Move	RNEXT, 4(RNEWREC)	
	TAIL	Move	RNEWREC, 4(RCURRENT)	
		Return		

92

图2-37 用于将新记录插入到一个链表中的子程序

### 删除一条记录

从一个链表中删除一条已存在的记录比插入一条新记录的操作要容易。我们连续简单地查询这个表，直到找到这个将要被删除记录的ID，然后进行所需要的链接字段的调整。图2-38给出了实现这种删除操作的子程序。假设寄存器RIDNUM中包含着将要被删除记录的ID。寄存器RHEAD、RCURRENT和RNEXT扮演着如同在插入子程序中同样的角色。第一个比较/转移指令检测将要被删除的记录是否是表头，如果是，将该头记录中的链接字段地址移入到RHEAD中，将该记录删除。要注意的是，如果这个头是该表中的惟一记录，则它的链接字段包含的内容是0，表示它同时也是表尾，所以将0移到RHEAD中正好表示空表的情况。如果头记录不是将要被删除的记录，则转移到SEARCH。这时，寄存器RCURRENT和RNEXT用来完成前向检索，从头开始直到所需要的记录被找到。当所需要的记录被第二个比较/转移指令对找到后，程序的执行将转到DELETE处。将这条被RNEXT指出的记录的链接字段改变为前一条记录的链接字段，即由RCURRENT指出的记录，这样这条记录就被删除了。最后两个Move指令实现了通过寄存器RTEMP的迁移。如果存储器到存储器的Move指令是允许的，则单指令

Move 4(RNEXT), 4(RCURRENT)

可以替换这两个Move指令。

93

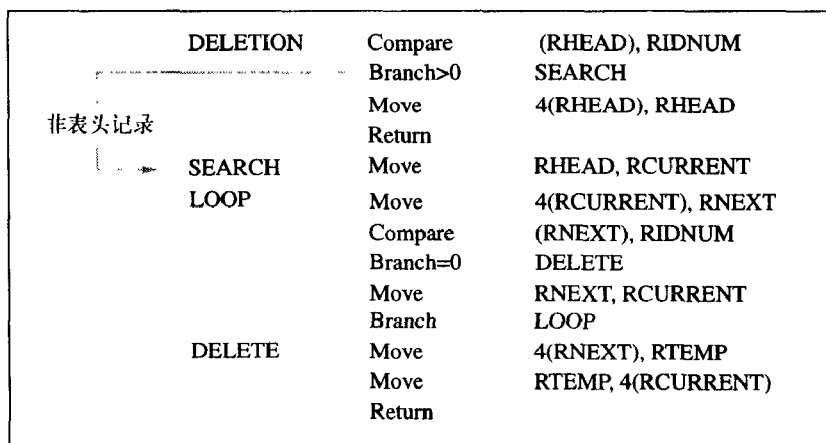


图2-38 从链表中删除一条记录的子程序

### 错误条件

在图2-37和图2-38中的插入和删除子程序没有考虑两种可能的错误条件，插入子程序可能碰到这样一种假定的情况，在列表中已经有一个与新记录的ID号相同的记录；而删除子程序可能会碰到在表中给出的ID号记录已经被删除。在习题2.23和2.24中考虑修改这些子程序，使它们可以处理这些可能发生的错误情况。

## 2.12 机器指令的编码

我们已经介绍了各种有用的指令和寻址方式，这些指令具体说明了由处理器电路为了执行这些具体任务时必须执行的动作，通常将它们称为机器指令。实际上，那些已经给出的指令格式是在汇编语言中使用的格式，不包括那些试图要避免使用的用于各种操作的缩写格式，那些操作是难以记忆的并且很可能是特殊的商用处理器所特有的。为了能够在一个处理器中执行，指令必须按照紧缩的二进制模式进行编码。这些编码后的指令才真正地称为机器指令。那些使用符号名和首字母缩写形成的指令叫做汇编语言指令，它们被汇编程序转换成机器指令，就像在2.6节中解释的那样。

在上一节中，我们做了一个简化处理的假设，即所有的指令是一个字长的。因为通常引用的是32位的字，这个假设意味着该长度对于所需要表示的信息是合适的。现在来考虑这种假设的有效性。

94

我们已经看到了执行像加、减、传送、移位、循环和转移这样操作的指令，这些指令可以使用不同大小的操作数，例如像32位和8位数字，或者8位ASCII编码字符。操作类型是将被执行的操作以及使用操作数的类型，它可能会用一个二进制模式的编码来表示，这个二进制编码称为该指令的操作码。如果分配了8位，就可以有256种用于说明不同指令的可能性。其余的24位说明所需信息的其他内容。

让我们来考察一些典型的情况，指令

Add R1, R2

除了操作码以外，还指出了寄存器R1和R2。如果处理器有16个寄存器，这时需要四位用来识别不同的寄存器。需要用附加位来说明该寄存器用在各种操作数中的寻址方式。

## 指令

Move 24(R0), R5

需要用16位来表示操作码和两个寄存器, 以及一些用来表示使用变址寻址方式的源操作数的位, 这里变址值是24。假设用三位来说明在表2-1中的一种寻址方式, 则用于这一目的就有6位, 它们代表源和目的操作数寻址方式选择, 因此左边的10位用来给出变址值。如果这10位足够用来表示用于变址的有符号数的给定范围, 则该指令可以放进32位的字中。

## 移位指令

LshiftR #2, R0

## 和数据传送指令

Move #3A, R1

除了用于说明操作码、寻址方式和寄存器的18位以外, 还必须分别说明立即数2和3A。这就将立即操作数的大小限定在14位的范围之内。

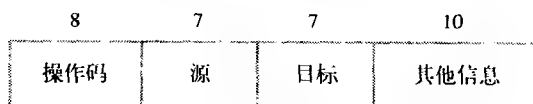
考虑以下的转移指令:

Branch > 0 LOOP

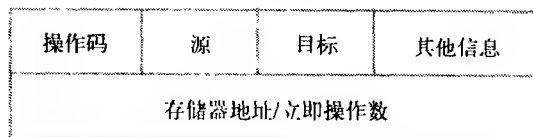
我们再一次用8位作为操作码, 其余的24位用来说明转移的偏移量。由于偏移量是二进制补码数, 转移目标地址必须是在转移指令所在位置的 $2^{23}$ 字节以内。为了转移到这个范围以外的一个指令上, 必须使用不同的寻址方式, 比如像直接寻址或是寄存器间接寻址等。使用这些方式的转移指令通常叫做跳转(Jump)指令。

在所有这些例子中, 指令可以在一个32位的字中进行编码。图2-39a描述了一种可能的形式。这里有一个8位的操作码字段以及两个7位的用来说明源和目的操作数的字段。7位字段用来识别寻址方式以及相关的寄存器(如果有的话)。“其他信息”字段可以用来指明那些可能需要的附加信息, 比如一个变址值或是一个立即操作数。

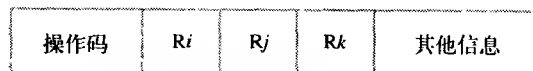
但是如果试图使用绝对寻址方式指明一个内存操作数时会出现什么情况呢? 指令



a) 单字指令



b) 双字指令



c) 三操作数指令

图2-39 32位字长的编码指令

Move R2, LOC

需要18位去说明操作码、寻址方式和寄存器。其余的14位用来表示与LOC相对应的地址, 这显然是不够的。如果我们想要能够在这个指令中给出一个完整的32位地址, 惟一的解决办法就是在这个指令中要包含第二个字, 在这种情况下, 添加的这个字可以保存需要的存储器地址。这种对应的形式已在图2-39b中给出。第一个字可能与图中的a部分相同, 第二个字完全是一个存储器地址。这种形式也可以适应于这样的指令:

And #FF000000, R2

在这种情况下, 第二个字给出了一个完整的32位的立即操作数。

如果我们想要允许在一个指令中的两个操作数都可以使用绝对寻址方式表示, 例如:

Move LOC1, LOC2

这时它就需要使用两个附加的32位地址的操作数。

这种方法产生了变量长度的指令, 其长度取决于操作数的个数和使用寻址方式的类型。使用多个字, 我们可以实现非常复杂的指令, 这非常类似于高级程序设计语言。复杂指令集计算机 (CISC) 可以用于访问那些使用这种指令系统类型的处理器。

96

这两种实现方法中存在着一个根本的不同之处, 如果我们坚持所有的指令必须安排到一个单一的32位字中, 就不可能在指令内部提供一个32位的地址或是一个32位的立即操作数。但仍然是有可能定义一个高性能的指令系统, 它可以大量地使用处理器寄存器。这样可以有

Add R1, R2

而不是

Add LOC, R2

作为后一个指令的替代, 我们可以使用

Add (R3), R2

只要在这条指令执行前将地址LOC装入到寄存器R3中即可。在这种情况下, 寄存器R3是作为指向所需存储单元的一个指针被使用的。

这就提出了一个问题, 如何将一个32位的地址装入到一个作为指向存储单元指针的寄存器中。一种可能是, 汇编程序直接将所需要的地址放到靠近程序的数据区中的一个字单元中。然后可以用相对寻址方式来装载这个地址。这里要假定在Load 指令中包含的变址字段足够大, 可以达到保存所需地址的位置上。另一个可能性是, 使用给出它足够小的一部分, 并用逻辑和移位指令去构造所需要的32位地址, 而这个小部分只要足够可以说明使用的立即寻址方式即可。这个问题在第3章对ARM处理器的描述中有更详细的阐述。所有的ARM指令都被编码成一个单独的32位字。

一个指令只占用惟一的一个字的限制已经引导了一种计算机风格, 它就是众所周知的精简指令集计算机 (RISC)。RISC方法引入了另一种限制, 比如所有的数据操作必须在操作数已经装入到处理器的寄存器中才能完成。这个限制意味着以上的加法指令将需要一个两指令的序列来完成:

Move (R3) R1

Add R1, R2

如果Add指令只能指明两个寄存器, 那么它将只需要32位字中的一部分。这样有可能提供一个更强大的使用三个操作数的指令

Add R1, R2, R3

它执行的操作是:

$R3 \leftarrow [R1] + [R2]$

对于这种指令的构成形式在图2-39c中给出。当然, 处理器必须能够处理这种三操作数的指令。在这样的指令集中, 所有的算术和逻辑操作只能使用寄存器操作数, 惟一与内存相关联的操作是将操作数加载到处理器的寄存器中或是向处理器寄存器中存储操作数。

97

RISC型的指令集与CISC型的指令集相比具有较少和较简单指令的特点。我们将在第8章中讨论RISC和CISC指令的相关特点，它与处理器设计的细节有关。

## 2.13 结束语

这一章以程序设计员的角度介绍了汇编与机器级的程序和指令的表示和执行。在讨论中强调了寻址技术和指令序列的基本原则。程序设计例子说明了使用某种现代计算机指令系统实现操作的基本类型。已经介绍的几种寻址方式包括了指针和变址寻址等重要概念。已经讨论的基本I/O操作给出了字符如何在处理器与键盘及处理器与显示设备之间进行传递的方法。对子程序的概念和实现子程序需要的指令也进行了讨论。子程序链接法提供了堆栈数据结构应用的一个例子，还介绍了机器指令控制操作其他数据结构的方法。对于队列、数组和链表也作了介绍。我们描述了两种不同机器指令集——CISC和RISC的设计方法。这两种设计方法的执行时间性能将在第8章中进一步描述。

## 习题

2.1 将十进制数5、-2、14、-10、26、-19、51和-43按以下的二进制方式表示成有符号的7位数字：

(a) 原码

(b) 反码

(c) 补码

(参见附录E“十进制整数到二进制整数的转换”。)

2.2 (a) 将以下各对十进制数转换成5位带符号的补码形式，并且将它们相加。判断在下列情况中是否有溢出发生。

(1) 5和10

(2) 7和13

(3) -14和11

(4) -5和7

(5) -3和-8

(6) -10和-13

(b) 重复(a)中的描述，对其做减法操作，即第一个数减去第二个数。查看在这些情况中是否有溢出发生。

2.3 在一些内存单元中给出一个二进制模式，是否能说明这些模式表示的是一个机器指令还是一个数字？

2.4 一个字节存储单元中包含着模式00101100。当作为一个二进制数解释时这个模式表示什么？作为一个ASCII码解释时它又表示什么？

2.5 考虑一台计算机，它有一个按字节寻址的存储器，按照32位字对应于big-endian策略组织方式。一个程序在键盘上读入ASCII字符并将它们存储在连续字节单元中，起始位置在1000处。在输入名字“Johnson”以后，显示存储在单元1000和1004中的两个字的内容。

2.6 针对little-endian策略重复问题2.5。

2.7 一个程序从键盘输入中读入表示十进制数的ASCII字符，并将这些字符存储在连续的内存字

节中。按照附录E的方式检查这些ASCII码并说明将每个字符转换成相等的二进制数时需要什么样的操作。

- 2.8 写一段程序，可以在一个单累加器的处理器中计算表达式

$$A \times B + C \times D$$

的值。假定该处理器有Load、Store、Multiply和Add指令，并且所有这些值都正好能放在该累加器中。

- 2.9 将图2-14中给出的学生成绩表改变为每个学生包含有 $j$ 项考试分数。假设这里有 $n$ 个学生。写一段汇编语言程序，计算在每项考试中的分数和，并且将这些和存储在存储器的字单元SUM、SUM+4、SUM+8、...中。考试的个数 $j$ 比处理器的寄存器可保存的数字要大，所以在图2.15中给出的针对于三项考试情况的典型程序已不能使用。使用两个嵌套循环，就像在2.5.3节中建议的那样，内部循环应该累计每个特定考试的和，而外部循环应该在考试数 $j$ 之上运行。假设 $j$ 存储在存储单元J中，放在单元N的前面。

- 2.10 (a) 针对一个指令集其中的算术和逻辑运算只适应于操作数在处理器寄存器中的情况，重写在图2-33中的点积程序。两个指令Load和Store被用来作为寄存器和存储器之间的操作数传递。

(b) 计算在表达式 $k_1 + k_{2n}$ 中的常数 $k_1$ 和 $k_2$ 的值，它表示执行(a)部分程序所需的存储器访问数量，其中包括指令字的读取。假设每个指令占用一个单独的字。

- 2.11 对于一个具有二地址指令的计算机，即它可以执行这样的操作：

$$A \leftarrow [A] + [B]$$

重复2.10中的问题，这里A和B可以是存储单元也可以是处理器寄存器。哪种计算机需要较少的内存访问？（第8章中的流水线给出了对于这个问题不同观点。）

99

- 2.12 “具有大量的处理器寄存器可以在执行复杂任务时减少访问内存的需求次数。”设计一个简单的计算任务，对一个有四个寄存器的处理器和另一个只有两个寄存器的处理器做比较，证明这句话的正确性。
- 2.13 在一个计算机中寄存器R1和R2保存着十进制值1200和4600，在以下的各指令中内存操作数的有效地址是多少？

- (a) Load 20(R1), R5
- (b) Move #3000, R5
- (c) Store R5, 30(R1, R2)
- (d) Add -(R2), R5
- (e) Subtract (R1) +, R5

- 2.14 假设由图2-14中给出的学生考试成绩表被当作一个链表存储在内存中，如图2-36所示。写一个汇编语言程序，它能完成像图2-15程序中一样的功能。头记录存储在内存单元1000处。
- 2.15 考虑包含有 $A(i, j)$ 的一个数组，这里 $i$ （从0到 $n-1$ ）是行标志， $j$ （从0到 $m-1$ ）是列标志。这个数组被一行接一行地存储在计算机的存储器中，它的每一行元素占有 $m$ 个连续的字单元。假设这个存储器是按字节寻址的，并且字长是32位。写一段将 $x$ 列加到 $y$ 列上的子程序，对应元素相加，将元素相加的和放在 $y$ 列上。 $x$ 和 $y$ 的标志被传送到子程序的R1和R2寄存器中，参数 $n$ 和 $m$ 被传递到子程序的R3和R4寄存器中，并且单元 $A(0, 0)$ 的地址被传递到寄存器R0中。

可以使用表2-1中的任何一种寻址方式。最多一个指令只有一个操作数可以在内存中。

2.16 以下两种语句都可以使值300存储在单元1000中，但是却是在不同的时间完成的。

```
ORIGIN      1000
DATAWORD    300
```

以及

```
Move #300, 1000
```

请解释这种差异。

2.17 在一个程序中寄存器R5用来指向一个栈的栈顶。使用变址、自动递增和自动递减寻址方式写一个指令序列，执行以下的各个任务：

100

- (a) 弹出栈顶上的两项内容，将它们相加，然后将结果压入栈中。
- (b) 从栈顶将第五项内容拷贝到寄存器R3中。
- (c) 从栈中删除前十项内容。

2.18 一个按字节的FIFO队列保存在内存中，它占有一个 $k$ 字节的固定范围。需要两个指针，一个IN指针和一个OUT指针。IN指针保存将要被添加到队列中的下一个字节所在位置轨迹，而OUT指针保存将从队列中移出的下一个字节所在位置的轨迹。

- (a) 当数据项加到队列中时，它们被加到连续的高地址处，直到达到存储范围顶端。当一个新的数据项加到队列时，下一步将发生什么事情？
- (b) 对于IN和OUT指针选择一个合适的定义，说明它们指向数据结构中的什么内容？使用简图说明你的答案。
- (c) 证明如果这个队列的状态只能用两个指针描述，则队列完全满和完全空这种情况就是难以区分的。
- (d) 为了解决(c)中的问题需要添加什么条件？
- (e) 针对操作IN和OUT两指针向该队列中添加和从该队列中移出数据项过程，给出一个程序。

2.19 考虑在问题2.18中描述的队列结构，写出APPEND和REMOVE程序，它们完成将数据在处理器寄存器和该队列中进行传送的任务。每当一个操作被检测并执行时，要小心地检查并修改队列和指针的状态。

2.20 为了保存一个子程序的返回地址，考虑以下情况：

- (a) 在处理器的寄存器中。
- (b) 在一个与调用有关的存储单元中，因此当子程序从不同的地方被调用时将使用不同的单元。
- (c) 在一个堆栈中。

在这些可能性中哪一个支持子程序嵌套？哪一个支持子程序递归（也就是子程序调用自身）？

2.21 计算机的子程序调用指令将返回地址保存在称为链接寄存器RL的处理器寄存器中，为了允许子程序能够嵌套将需要做什么？将采用什么策略使得子程序可以调用自身？

2.22 假设想要在一个计算机中按如下方式组织子程序调用：当程序Main想要调用子程序SUB1时，它调用一个中间程序CALLSUB，并且将SUB1的地址作为一个参数传递给它，放在寄存器R1中。CALLSUB将返回地址保存到一个栈中，并确认该栈的上限没有超出。然后转移到SUB1中。为了返回到调用程序中，子程序SUB1调用另一个中间程序RETRN。这个程

101



序检测到这个栈是非空的，然后使用栈顶元素返回到原调用程序中。

写出程序CALLSUB和RETRN，假定子程序调用指令将返回地址保存在一个链接寄存器RL中，这个栈的上限和下限分别记录在内存单元UPPERLIMIT和LOWERLIMIT中。

- 2.23 在图2-37的链表插入子程序中，没有检测匹配的新记录ID号是否在该列表中已存在一条记录。在子程序执行中如果是这种情况，将会发生什么事情？修改这个子程序，如果出现这种情况，将匹配记录的地址返回到寄存器ERROR中；如果插入成功返回一个0。
- 2.24 在图2-38的链表删除子程序中，已假定保存在寄存器RIDNUM中ID号的记录在表中是存在的。如果在这个子程序执行中没有该ID号的记录时将会发生什么情况？修改这个子程序，如果删除成功给RIDNUM返回一个0；如果记录没有在这个表中，则保持RIDNUM的值不变。

# ARM、Motorola与Intel指令集

## 本章目标

本章由三个独立的部分组成。你将了解到以下指令集（指令系统）体系结构：

- ARM（部分I）
- Motorola 68000（部分II）
- Intel IA-32 Pentium（部分III）

103

在第2章中我们已经介绍了指令集、寻址方式以及指令执行的基本概念，并采用汇编语言编写了一些机器指令和程序作为例子进行了说明。本章将学习这些基本概念是如何在ARM、Motorola 68000和Intel IA-32上实现的。其中，ARM指令集是采用RISC设计方式的典型例子，而68000和IA-32则采用了CISC设计方式。本章由三部分构成，三种指令集各占一部分，每一部分都是完全独立的单元。为了能更好地理解第2章中的基本概念和程序，我们在本章中采用更加简要的方式进行讨论。在附录B、附录C和附录D中给出了这三种ISA的概述，其中的内容比本章要更详细。

## 部分I ARM实例

Advanced RISC Machines（ARM）有限公司已经设计出了一系列微处理器，同时还允许将此设计用于其他公司的计算机产品和嵌入式系统的芯片应用中。ARM公司是一个比较年轻的公司，正处于不断发展的阶段，它的前身是20世纪80年代早期主要开发处理器的Acorn Computers公司。ARM微处理器主要用于低功耗、低成本的嵌入式产品，如移动电话、通信中的调制解调器、自动机车管理系统以及掌上数字助手<sup>[1]</sup>。Furber<sup>[2]</sup>的著作对于ARM设计和实现作了详细阐述；在Clements<sup>[3]</sup>的书中使用ARM作为主要实例，在van Someren和Attack<sup>[4]</sup>的书采用汇编语言对ARM进行了描述。我们还可以从ARM的网站<sup>[5]</sup>上获取更多详细信息。所有的ARM处理器都具有相同的基本机器指令集。本章使用的是在ARM7处理器中实现的版本。后续版本增加了一些与本章讨论的内容不相关的其他特征，在第11章中将介绍这些增加的特征。为了说明ARM体系结构的不同特点，我们采用第2章中的程序进行讨论，并且用ARM汇编语言来说明ARM体系结构的各个方面。

## 3.1 寄存器、内存访问及数据传递

在ARM体系结构中，内存是按照字节进行编址的，每个字节的地址是32位，处理器中寄存器的长度也是32位。在内存和处理器寄存器之间传送数据时采用的是两种操作数长度：字节（8位）和字（32位）。字地址必须是对齐的，也就是必须是4的倍数，且同时支持little-endian和

big-endian两种内存寻址方式（参见2.2.2节）。两种寻址方式的选择是通过一条与处理器相连的外部输入控制线来完成。当内存向处理器寄存器载入字节或寄存器向内存存储字节时，通常使用寄存器的低位部分。

内存访问只使用Load和Store指令。所有的算术和逻辑指令操作的数据都放在处理器寄存器中。这是RISC体系结构的基本特征。这种处理器设计的简化意义及其性能我们将在第8章中介绍。

### 3.1.1 寄存器结构

应用程序使用的处理器寄存器如图3-1所示。图中一共有16个32位寄存器，分别标注为R0到R15，这些寄存器中包含了15个通用寄存器（R0到R14）和一个程序计数器（PC）寄存器R15。通用寄存器可以保存任何一个内存地址或操作数。当前程序状态寄存器（CPSR）或简单的状态寄存器中，保存的内容是条件码标志（N, Z, C, V）、中断禁止标志和处理器模式位。条件码标志代表的信息我们在2.4.6节中描述过。处理器模式位和中断禁止标志的使用，我们将在第4章中的输入/输出操作和中断的相关部分中进行描述。此处，我们假定处理器的工作模式是用户模式，并且正在执行一个应用程序。

此外，还有15个附加的通用寄存器，称为后备寄存器。它们保存的内容是R0到R14寄存器中的某些副本。当处理器切换到管态或中断操作方式的时候就要使用后备寄存器了。这些后备寄存器和状态寄存器副本内容也将在第4章中介绍。

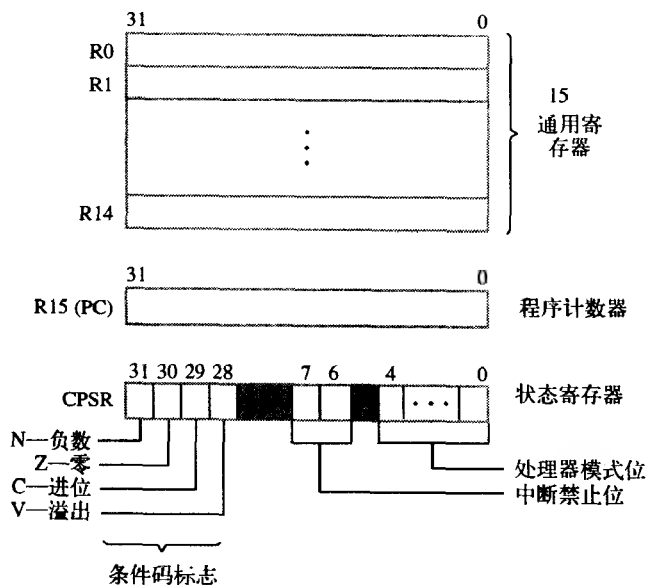


图3-1 ARM寄存器结构

### 3.1.2 内存访问指令和寻址方式

ARM体系结构中的每条指令都是采用典型的RISC设计方式，按照相应的统一的方式编码成一个32位的字。内存访问只提供了两条指令：Load和Store指令。这些指令的基本编码格式与传送指令、算术指令及逻辑指令一样，都采用图3-2所示的格式。附录B中给出了更详细的说明。在一条指令中指定了一个条件执行码（条件）、OP码、两个或三个寄存器（Rn, Rd和Rm），以及

一些其他信息。如果不需要使用寄存器*Rm*，则“其他信息”字段就扩展到*b*<sub>0</sub>位。在Load指令中，操作数从内存传送到通用寄存器中，该寄存器使用4位的*Rd*字段来命名。在Store指令中，操作数从*Rd*传送到内存中。如果操作数是字节，通常将其放在寄存器的低位字节部分，在Load指令中，寄存器中高位的24位全部用0填充。

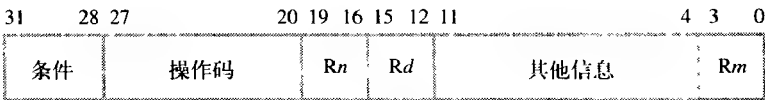


图3-2 ARM指令格式

指令的条件执行

ARM处理器的特色也是很特别的，即全部的指令都是依赖于在指令中指定的条件而完成的条件执行。只有处理器条件码标志的当前状态满足了指令中的位*b*<sub>31-28</sub>所指定的条件时，该指令才可以执行。否则，处理器将处理下一条指令。其中有一个条件是用来表示该指令总是被执行的。条件执行的优点我们将在3.7节中通过例子来说明。下面，暂时忽略这个特点并假定指令中的条件字段中的代码是“总被执行”的。

内存寻址方式

对内存操作数寻址的基本方法就是产生操作数的有效地址（EA），也就是通过将基址寄存器*Rn*的内容加上一个带符号的偏移量产生的结果，其中的*Rn*是在图3-2中所示指令中指定的。偏移量的值可以是包含在指令中最低12位中的一个立即数，或者是第三个寄存器（*Rm*，用指令中的最低四位（*b*<sub>3-0</sub>）所命名）的内容。偏移量的符号（方向）包含在操作码字段中。

例如，Load指令

```
LDR Rd, [ Rn, # offset ]
```

指明在立即数方式中的偏移量（表示为一个有符号数），执行的操作是

$$Rd \leftarrow [ [ Rn ] + offset ]$$

注意：目标寄存器*Rd*要放在前面。这种用法与第2章中的用法正好相反。指令

```
LDR Rd, [ Rn, Rm ]
```

执行的操作是

$$Rd \leftarrow [ [ Rn ] + [ Rm ] ]$$

由于*Rm*中的内容是偏移量的值，如果需要的是负数偏移量，必须在*Rm*前面带一个负号。在第2章中，这两种寻址方式分别对应于变址方式中的变址和基址。如果偏移量是0就无需明确指定。因此，指令

```
LDR Rd, [ Rn ]
```

执行的操作是

$$Rd \leftarrow [ [ Rn ] ]$$

所使用的寻址方式就是第2章中的间接寻址方式。

操作码助记符 LDR 表示将一个32位的字从内存装入到寄存器中。如果是字节操作数，就使用LDRB将该操作数放到寄存器的低字节位置。高字节部分全部用0填充。

Store指令的助记符有STR和STRB。例如，指令

STR Rd, [Rn]

执行的操作是

$[Rn] \leftarrow [Rd]$

将一个字操作数传送到内存中。STRB指令传送的是Rd中低位部分中的字节。

ARM文档中提到的所有这些模式以及其他将要简要描述的模式都采用基于变址的寻址方式。在前面的例子中所使用的形式称为预变址寻址方式，因为操作数的有效地址是通过将基址寄存器Rn的内容加上偏移量来产生的。其中基址寄存器Rn的内容是不变的。这种寻址方式与我们第2章中讨论过的自动递减和自动递增方法相类似。它们分别称为“带写回的预变址”和“传递变址”。

这三种方法的定义如下：

预变址方式——操作数的有效地址是基址寄存器的内容与偏移量的和。

带写回的预变址方式——操作数的有效地址的产生方法与预变址方式相同，之后，有效地址要写回到Rn中。

传递变址方式——操作数的有效地址是Rn的内容。偏移量然后加到这个地址上，并且这个结果写回到Rn中。

表3-1说明了这些寻址方式的汇编语言语法，并给出了计算有效地址（EA）和写回操作的表达式。在预变址寻址方式中的惊叹号是用来表示写回操作的。由于传递变址方式中经常包括写回操作，因此将惊叹号省略了。注意，要区别预寻址和传递寻址中方括号的不同使用方法。当方括号中只有基址寄存器时，该寄存器的内容当作有效地址使用。在访问完操作数之后，将偏移量与寄存器的内容相加。换句话说，指定的是传递变址方式。它是在2.5节中描述过的自动递增寻址方式的一种概括形式。当基址寄存器和偏移量同时放到方括号中时，那么操作数的有效地址就是两者之和，也就是使用预变址方式。如果要执行回写操作，必须用惊叹号（!）来标明。带有写回的预变址是在2.5节中讨论过的自动递减寻址方式的一种概括形式。

表3-1 ARM变址寻址方式

名 称	汇 编 语 法	寻 址 功 能
带有立即偏移量 预变址	[Rn, #offset]	EA = [Rn] + offset
带写回的预变址	[Rn, #offset]!	EA = [Rn] + offset Rn ← [Rn] + offset
传递变址	[Rn], #offset	EA = [Rn] Rn ← [Rn] + offset
Rm 中带有偏移量值 预变址	[Rn, ± Rm, shift]	EA = [Rn] ± [Rm] shifted
带写回的预变址	[Rn, ± Rm, shift]!	EA = [Rn] ± [Rm] shifted Rn ← [Rn] ± [Rm] shifted
传递变址	[Rn], ± Rm, shift	EA = [Rn] Rn ← [Rn] ± [Rm] shifted
相对的寻址方式 带有立即偏移量的预变址	位置	EA = Location = [PC] + offset

EA = 有效地址

Offset = 包含在指令中的有符号数

Shift = 方向 # 整数

这里方向是LSL时为左移，是LSR时为右移；整数是一个5位的无符号整数，用来说明移位的量

± Rm = 寄存器Rm中的偏移量值，可以从基址寄存器Rn中加上或减去

在所有这三种寻址方式中，可以将偏移量指定为范围在 $\pm 4095$ 之间的一个立即数。同时，偏移量的值还可以通过第三个寄存器（ $R_m$ ）的内容来指定，在该寄存器名称前面带有 $\pm$ 前缀的偏移量符号（方向）。例如，指令

LDR R0, [R1, -R2]!

108

执行的操作是

$R0 \leftarrow [R1] - [R2]$

由于用惊叹号说明了写回，所以操作数 $[R1] - [R2]$ 的有效地址然后被装入到R1中。

如果偏移量采用寄存器指定，那么它可以通过向左向右移位缩放2的乘幂。这在汇编语言中是在寄存器 $R_m$ 之后放置指定移动方向（LSL是向左移动，LSR是向右移动）和移动的位数来实现的，如表3-1所示。移动的位数是通过范围在0到31之间的一个立即数指定。例如，上例中的R2的内容在作为偏移量使用之前可能要乘以16，那么将它表述如下：

LDR R0, [R1, -R2, LSL #4]!

该指令将执行如下操作

$R0 \leftarrow [R1] - 16 \times [R2]$

之后，将有效地址装到R1中。

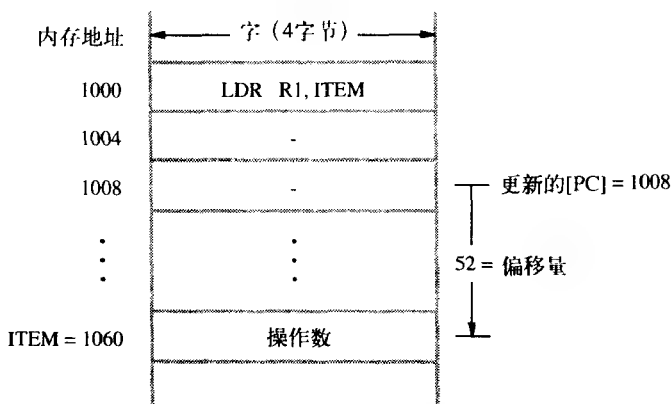
程序计数器（PC）可以当作基址寄存器 $R_n$ 使用。这种情况下所使用的是在2.5节中描述过的相对寻址方式。汇编程序要判定在操作数地址和更新过的PC内容之间的一个有符号的距离，将它作为一个立即偏移量。当有效地址是在指令执行期间被计算出来的，那么PC内容已经更新到从包含有该相对寻址方式指令向前的两个字（8个字节）的地址上。这样做的原因是与指令的流水线执行有关，我们将在第8章中介绍。

图3-3a中给出了相对寻址方式的一个例子。指令中给出的象征符号ITEM表示操作数的有效地址，它的值是1060。在ARM体系结构中不能使用绝对寻址方式。因此，当用汇编语言按照这种方式给出操作数的有效地址时，汇编程序通常使用相对寻址方式。它采用的是带有立即数的偏移量，采用将PC作为基址寄存器的预变址方式来实现的。正如图中所示，汇编程序计算出来的偏移量是52，由于PC在程序执行期间将偏移量加到PC上的时候它的值是1008，因此生成的有效地址是 $1060 = 1008 + 52$ 。操作数必须处于被更新的PC值之前或之后的4095的范围内。如果指令中给出的操作数的地址超出了这个范围，那么汇编程序将显示错误，并采用其他的寻址方式来访问该操作数。

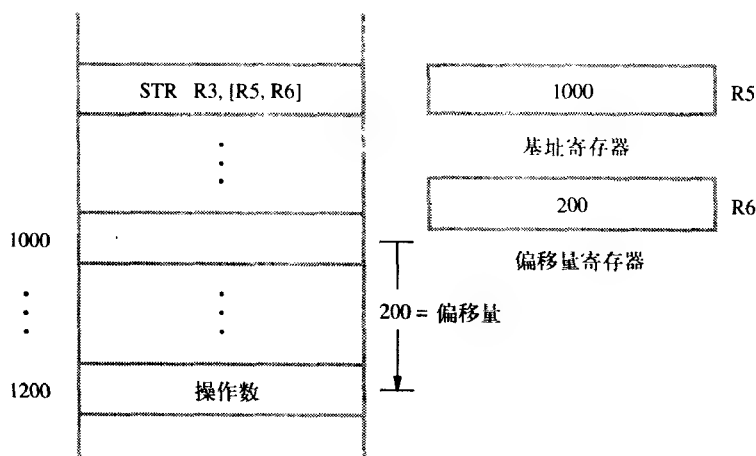
图3-3b给出的是预变址方式的例子，该方式中的偏移量放在寄存器R6中，基址的值放在R5中。Store指令（STR）将R3中的内容保存到内存1200单元的字中。

图3-4中所示的例子说明的是在传递变址和预变址寻址方式中的写回特征的有效性。图3-4a显示的是具有25个数，每个数分别占用25个字的列表中的前三个数，表的起始内存地址是1000。这样，它们构成了一个 $25 \times 25$ 按列存储的数值矩阵的第一行。矩阵中的第一行的第一个数保存在位置为1000的字中。第一行的后续数分别保存在地址1100, 1200, ..., 3400处。25个内存位置1000, 1004, 1008, ..., 1096包含矩阵的第一列。

109



a) 相对寻址方式



b) 预变址寻址方式

图3-3 ARM内存寻址方式实例

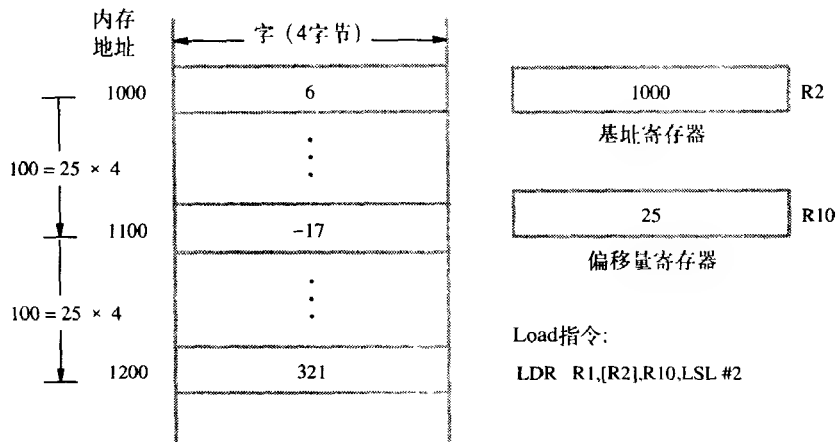
采用偏移量的带有写回的传递变址寻址方式，可以非常方便地对矩阵中的第一行连续的数值进行访问。假定R2用作基址寄存器，它的初始地址值是1000。寄存器R10用来保存偏移量，它被装入的值是25。则指令

110

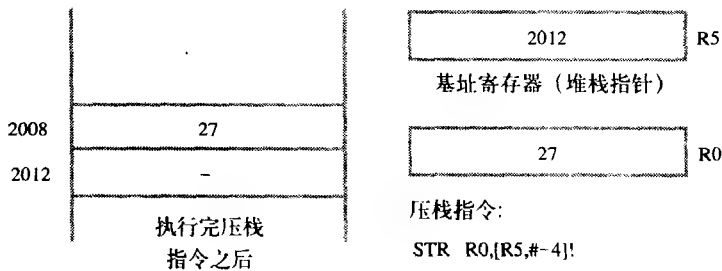
LDR R1, [R2], R10, LSL #2

可以用在一个通过连续的循环操作将矩阵第一行中的连续元素装入到寄存器R1中的程序中。下面让我们一步一步分析它是如何工作的。在第一次执行Load指令时，有效地址是[R2] = 1000。因此，该地址中的数值6被装到R1中。之后，写回操作将R2的内容从1000改写成1100，这样指向第二个数值-17。这是通过对偏移量寄存器中的内容25进行左移两位之后加上R2的内容而得到的。在这个处理过程中R10的内容是不变的。左移两位等价于25乘以4，它生成了所需的偏移量100。这个偏移量和R2的内容相加之后，新的地址1100被写回到R2中。在第二遍循环中执行Load指令时，第二个数-17被装到R1中。在第三遍中第三个数321被装到R1中，依次类推。

111



a) 带写回的传递变址寻址



b) 带写回的预变址寻址

图3-4 带有写回的ARM内存寻址方式

本例中包括将偏移量寄存器中的内容进行移位并与基址寄存器中的内容相加。像表3-1中说明的那样，移位后的偏移量还可以与基址寄存器的内容相减。可以选择的移动距离的范围是0到31，而且左右移动都可以。

图3-4b所示的是将寄存器R0中的内容27放到堆栈上的一个例子。寄存器R5被当作堆栈指针使用。最初，它包含当前TOS（栈顶）元素的地址2012。采用带有写回的预变址寻址方式，使用一个立即数偏移量来完成压栈操作，其指令是：

```
STR R0,[R5,#-4]!
```

立即数偏移量-4加到R5的内容2012上，之后将其写回到R5中。这个新TOS位置2008用作Store操作的有效地址。寄存器R0的内容27被存储到2008单元中。

多个操作数的Load/Store指令

除了单操作数的Load和Store指令以外，还有多操作数的Load和Store指令。它们称为“块传递指令”。任何通用寄存器都可以用来装入和存储。但是只允许字操作数，其操作码为LDM（装入多个）和STM（存储多个）。内存操作数的存储位置必须是连续的字单元。无论是预变址还是传递变址方式，带写回和不带写回的寻址方式都可以采用。它们使用的基址寄存器Rn是在指令中指定的。在这些指令中偏移量的值通常都是4，所以无需在指令中明确指明。在汇编语言表达式中，指令中使用的寄存器列表必须是按照升序出现。例如，假定寄存器R10用作基址寄存器，



它的初始值是1000。那么，指令

**LDMIA R10!, {R0, R1, R6, R7}**

将位置1000, 1004, 1008和1012处的字传送到寄存器R0, R1, R6, R7中，在最后的传送完成之前，地址1016一直保存在R10中。操作码中的后缀IA表示“后增”，对应于传递变址。我们将在后面的3.6节中结合子程序的实现（有关在堆栈上保存和恢复寄存器的有效方式）来讨论Load/Store多操作数指令。

### 3.1.3 寄存器传送指令

经常需要将一个寄存器中的内容复制到另一个寄存器中，或者是将一个立即数存放到一个寄存器中。传送指令

**MOV Rd, Rm**

使用图3-2中的指令格式，将寄存器Rm中的内容复制到寄存器Rd中。指令中低8位的立即数操作数通常也是利用Move指令来装入到寄存器Rd中的。例如，

**MOV R0, #76**

将立即数76放到寄存器R0中。在这两种传送指令形式中，原操作数在放到目标寄存器之前都是可以进行移位操作的。

## 3.2 算术和逻辑指令

ARM指令集中有很多用于算术和逻辑操作的指令，其操作数可以是在通用寄存器中，也可以是指令本身中给出的立即操作数。这些指令不允许使用内存操作数。加法和减法有不同形式的指令，此外还有两条乘法指令，以及一些逻辑指令（如与、或、非、异或和位清除）。指令集中还提供了基于两个操作数的算术或逻辑操作的结果来设置条件码标志的比较指令。比较指令并不将实际的结果保存到寄存器中。大部分这些指令的格式都如图3-2所示的那样。

### 3.2.1 算术指令

算术指令的汇编语言表达式是

操作码 **Rd, Rn, Rm**

由操作码指定的操作是利用通用寄存器Rn和Rm来完成的，结果保存到寄存器Rd中。例如，指令

**ADD R0, R2, R4**

执行的操作

**R0 ← [R2] + [R4]**

而指令

**SUB R0, R6, R5**

执行操作

**R0 ← [R6] - [R5]**

如果不使用寄存器Rm，第二个操作数可以用立即数的方式给出。这样

ADD R0, R3, #17

执行的操作

$R0 \leftarrow [R3] + 17$

立即数包含在指令中的低8位 $b_7 \sim b_0$ 中。

第二个操作数在用于操作之前可以进行移位或循环移位操作。当需要进行移位或循环移位操作时，在指令的汇编语言表达式的最后进行说明。指令

ADD R0, R1, R5, LSL #4

操作如下：在寄存器R5中的第二个操作数要左移4位（等价于 $[R5] \times 16$ ），然后与寄存器R1中的内容相加，和保存在寄存器R0中。

乘法指令提供两种版本，第一种版本将两个寄存器中的内容相乘，然后将乘积的低32位保存到第三个寄存器中，乘积的高位（如果有的话）将被丢弃。例如，指令

MUL R0, R1, R2

执行的操作

$R0 \leftarrow [R1] \times [R2]$

第二个版本指定了第四个寄存器，在将结果保存到目标寄存器中之前，它的内容与乘积相加。因此，指令

MLA R0, R1, R2, R3

执行操作

$R0 \leftarrow [R1] \times [R2] + [R3]$

这称为“乘法累加”操作。这种操作经常用于数字信号处理中的数字算法。我们将在3.7节中看到这类应用的一个例子。在图3-2中的第四个寄存器是在其他信息字段中进行编码的。在这两个操作数用于两条乘法指令之前，并没提供任何用于移位或循环移位的操作数。ARM ISA的一些版本中还允许双倍长度的乘积（64位）（请参见第11章）。

#### 操作数移位操作

我们注意到前面ARM指令系统的一个最显著的特征是它的全部指令都是有条件执行的。该系统的另一个特征是大多数的指令中都包含了移位操作和循环移位操作。而在其他的计算机指令系统中，移位操作都是采用单独的指令来完成的。在本章中的部分II和部分III中描述的Motorola 68000和Intel IA-32处理器就是采用这种方式的。将移位和循环移位操作按照需要集成到指令中，使得ARM体系结构可以节省一定的代码空间，并且可以潜在提高在执行时间方面的性能，从而更加方便了处理器的设计。这个特征是通过在处理器中的寄存器和算术逻辑单元之间的数据通路中称为桶形移位器电路来实现的。移位和循环移位的更多可用的细节以及它们指令格式编码部分，请参见附录B。

114

### 3.2.2 逻辑指令

逻辑指令与、或、异或和位清除所对应的操作码分别训AND、ORR、EOR和BIC。这些指令与算术指令具有相同的指令格式。指令

AND Rd, Rn, Rm

执行操作

$$Rd \leftarrow [Rn] \wedge [Rm]$$

它寄存器 $Rn$ 和 $Rm$ 中的操作数按位进行逻辑与操作。例如，如果寄存器R0包含的十六进制数是02FA62CA，R1的内容是0000FFFF，那么指令

AND R0, R0, R1

的结果是000062CA，并且结果被放到寄存器R0中。

位清除指令（BIC）与AND指令非常相近。该指令首先将寄存器 $Rm$ 中的内容按位求反之后，再将 $Rn$ 和 $Rm$ 中的内容按位进行与操作。还是采用上例中相同的R0和R1的位模式，指令

BIC R0, R0, R1

结果的位模式为02FA0000，并被放到R0中。

传送负数指令（操作码的助记符MVN）将源操作数的各位求反之后将结果放到 $Rd$ 中。如果R3的内容十六进制模式是0F0F0F0F，那么指令

115

MVN R0, R3

将结果F0F0F0F0放到R0中。

#### 数字打包程序

图3-5给出了一个将两个4位十进制数打包存放在一个字节单元中的ARM程序。这个程序的普通版本如图2-31中所示，并且在2.10.2节中已经描述过了。用ASCII表示的十进制数存储在字节单元LOC和LOC+1中。该程序将相应的4位BCD码打包成一个单字节并存放在PACKED处。

LDR	R0, POINTER	将地址LOC装入R0
LDRB	R1, [R0]	将ASCII字符装入R1和R2
LDRB	R2, [R0, #1]	
AND	R2, R2, #&F	将R2的高28位清除
ORR	R2, R2, R1, LSL #4	R1左移之后与R2进行或运算并将结果放入R2中
STRB	R2, PACKED	将打包的BCD数存到PACKED中

图3-5 将两个4位的十进制数打包到一个字节单元的ARM程序

图3-5程序中的第一个Load指令假定地址LOC已经存储到内存地址指针POINTER中。就像我们在3.4节中看到的一样，可以使用汇编程序指示符将LOC放到POINTER中。必须使用这种方法来将地址LOC装入R0中，因为在指令中不能使用32位的地址作为立即数。POINTER指向BCD码连续存放的字节处。在接下来的两个Load指令中，分别将两个ASCII码中保存的BCD数字（字符中最低4位）装入到寄存器R1和R2的低位部分。AND指令将R2的高28位清成0，保留最低四位中的第二个BCD数字。之后的或指令将R1中的第一个BCD数左移四位，并将其放到R2中第二个BCD数的左边。最后，R2中低字节打包后的数字就可以保存到PACKED中了。

### 3.3 转移指令

条件转移指令中包含着一个有符号的24位的偏移量，它是用来与程序计数器的更新内容相加以生成转移目标地址的。转移指令的格式如图3-6a所示，在图3-6b中给出了一个例子。指令BEQ（如果等于0就转移）在Z标志设置成1时便会产生转移。

需要检测在指令字的最高四位 $b_{31-28}$ 中的条件，以便判断是否需要产生转移。转移指令与其

他的ARM指令一样按照相同的方式执行，也就是，只有条件码标志的状态与相应的指令中条件字段的条件相一致时，转移才会执行。

在计算转移目标地址的同时，PC的内容已经更新成转移指令所在地址的下两个字的指令地址。如果转移指令的地址是1000，转移目标地址是1100，如图3-6b所示，那么由于在计算地址1100时，更新的PC内容将是 $1000 + 8 = 1008$ ，所以偏移量应该是92。

116

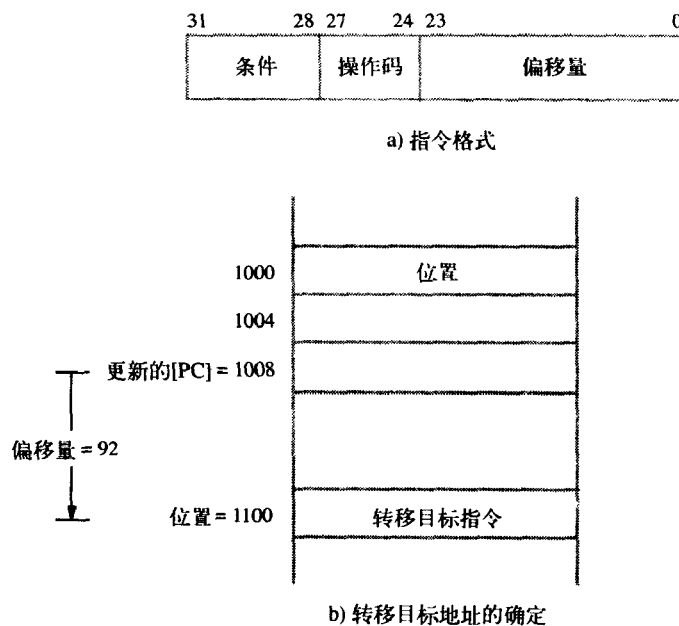


图3-6 ARM转移指令

### 3.3.1 设置条件码

有些指令如比较指令，如下所示：

CMP Rn, Rm

执行这样的操作：

$[Rn] - [Rm]$

它的惟一目的就是根据减法操作的结果来设置条件码标志。另一方面，算术和逻辑指令只有在操作码字段中的位设置明确说明时，才能影响条件码标志。这种方式通过在汇编语言操作码助记符后加上后缀S来表示。例如指令

117

ADDS R0, R1, R2

是要设置条件码标志的，而

ADD R0, R1, R2

就不设置条件码标志。

### 3.3.2 用于数值相加的循环程序

图3-7给出了列表中数值相加的循环程序，它模仿图2-16中的程序编写。前两步首先执行装

入和存储操作，最后的指令使用的是相对寻址方式。假定内存中N、POINTER和SUM都在与PC相关的偏移量的有效范围内。内存单元POINTER中的地址是第一个相加数NUM1的地址，N中存放的是列表中数据的个数，SUM用来保存相加的和。循环中第一条指令的带有写回的传递变址寻址方式是图2-16中自动递增寻址方式的真实反映。

	LDR	R1,N	将计数值装入R1
	LDR	R2,POINTER	将地址NUM1装入R2
	MOV	R0,#0	清空累加器R0
LOOP	LDR	R3,[R2],#4	将下一个数装入R3
	ADD	R0,R0,R3	相加数值到R0
	SUBS	R1,R1,#1	对循环计数器R1减1
	BGT	LOOP	如果没有完成，转移返回
	STR	R0,SUM	保存累加和

图3-7 用于数值相加的ARM程序

### 3.4 汇编语言

ARM汇编语言利用汇编指示保护存储空间、对地址标号和常数标识符分配数值、定义程序和数据块在内存中放置的位置以及指明源程序文本的结束位置。这些在2.6.1节中已经描述过了。

118 图3-8通过给出图3-7中程序的一个完整的源程序来列举一些ARM的汇编指令。AREA指示符使用的参数是CODE或DATA，表示包含程序指令或数据的内存块的开始。AREA指示符实际上需要更多的说明信息，但是这些与我们讨论的内容无关。ENTRY指示符说明程序的执行是从LDR指令开始的。

	内存地址 标号	操作	地址或 数据信息
汇编指示		AREA ENTRY	CODE
产生机器指令 的语句	LOOP	LDR LDR MOV LDR ADD SUBS BGT STR	R1,N R2,POINTER R0,#0 R3,[R2],#4 R0,R0,R3 R1,R1,#1 LOOP R0,SUM
汇编指示	SUM N POINTER NUM1	AREA DCD DCD DCD DCD END	DATA 0 5 NUM1 3, - 17, 27, - 12, 322

图3-8 图3-7中程序的ARM汇编语言源程序

在紧跟代码区之后的数据区中，DCD指示符是用于标注和初始化操作数的。SUM和N处的字在前两个DCD指示符处被分别初始化成0和5。地址NUM1利用一个DCD指示符放到POINTER指针处。最后的DCD指示符将相加的五个数放到以NUM1开始的连续内存位置中。

常量用十六进制符号表示时带有&前缀和基数 $n$ 。对于 $n$ 在2到9时，都表示成 $n\_xxx$ 。例如，2\_101100表示一个二进制常量。基数是10的常量不必带有前缀。

EQU指示符用于定义常量的符号名称。例如，语句

TEN EQU 10

119

允许TEN代替十进制常数10在程序中使用。当在程序中使用一个寄存器号时，利用它们的符号名称是非常方便的。RN指示符就是为此目的而设置的。例如

COUNTER RN 3

为寄存器R3建立了COUNTER名称。寄存器名称R0到R15、PC (R15) 以及LR (R14) 都可以由汇编程序进行预定义。

### 伪指令

将图3-8中地址NUM1装入到寄存器R2中的另一种方法是可以用汇编语言来实现。伪指令

ADR Rd, ADDRESS

将32位ADDRESS值装入到Rd中。该指令并不是真正的机器指令。汇编程序会选择适当的机器指令来实现伪指令。例如，在图3-8中使用的机器指令

LDR R2, POINTER

和数据声明指示

POINTER DCD NUM1

的结合就是实现伪指令

ADR R2, NUM1

的一种方式，它将会在程序中替换LDR指令中的相应位置。这种情况下，汇编程序将需要为DCD的声明分配一个合适的数据区。

在这个特殊的例子中实现ADR指令可能有更有效的方式，这将是 by 汇编程序来选择的。在ADR指令装入地址值时，PC (R15) 的当前内容不能超过255个字节，指令

ADD Rd, R15, #offset

可以用来实现ADR伪指令。如果在这个实例程序中这样做的话，则不需要POINTER单元。汇编程序使用真正的机器指令来实现ADR伪指令：

ADD R2, R15, #28

由于当执行ADD指令时的NUM1是28个字节，超出了更新后的PC值，这里假定数据区紧跟在STR指令的后面。实际上并不是这样的，因为STR指令后面跟着的是将控制返回给操作系统的指令，但在这里将其忽略了。

120

### 3.5 I/O操作

ARM体系结构采用的是在2.7节中描述过的存储器映射I/O。正如在本节中介绍的，从键盘读

取一个字符或向显示器发送一个字符，都可以通过程序控制I/O来完成。

假定设备状态寄存器INSTATUS（键盘）和OUTSTATUS（显示器）中的第三位分别包含着控制标志SIN和SOUT；同时假定键盘DATAIN和显示器DATAOUT寄存器紧跟在状态寄存器的后面，位于地址INSTATUS+4和OUTSTATUS+4处；地址INSTATUS已经装入寄存器R1中；那么读和写等待循环按照如下方式执行。指令序列：

```

READWAIT  LDR  R3,[R1]
           TST  R3,#8
           BEQ  READWAIT
           LDRB R3,[R1,#4]

```

当键盘上有一个键被按下时将该字符读入到寄存器R3中。测试（TST）指令将它的两个操作数进行按位逻辑与操作，并根据结果来设置条件码标志。立即数8在第三位上是1。因此，如果INSTATUS的第三位是0，则TST的操作结果将是0；如果第三位是1时，结果将是非零，这就意味着在DATAIN中有一个字符存在。对于BEQ指令，如果结果是0，则转移返回到READWAIT，一直循环直到有按键被按下将INSTATUS的第三位设置为1时为止。

假定地址OUTSTATUS已经被装入到寄存器R2中，指令序列

```

WRITEWAIT LDR  R4,[R2]
           TST  R4,#8
           BEQ  WRITEWAIT
           STRB R3,[R2,#4]

```

当显示器准备接受字符时将寄存器R3中的字符发送到DATAOUT寄存器中。

这两个程序可以从键盘读取一行字符，将它们存储到内存中，再将它们显示在显示器上，如图3-9中程序所示。该程序是图2-20中程序的模仿程序。假设寄存器R0中保存该行字符在内存中存储的第一个字节地址，寄存器R1到R4与上面描述的READWAIT和WRITEWAIT循环中的用法相同。第一个存储指令（STRB）将从键盘读入的字符存储到内存中。该指令中采用带有写回的传递变址寻址方式来遍历内存区，这类似于图2-20中的自动递增寻址方式的用法。测试是否相等的指令（TEQ）用来检测两个操作数是否相等，并设置Z条件码相应的标志。

121

READ	LDR	R3,[R1]	载入[INSTATUS]并等待
	TST	R3,#8	字符
	BEQ	READ	
	LDRB	R3,[R1,#4]	读取字符并将其保存在内存中
ECHO	STRB	R3,[R0],#1	
	LDR	R4,[R2]	载入[OUTSTATUS]并等待显示器就绪
	TST	R4,#8	
	BEQ	ECHO	
	STRB	R3,[R2,#4]	向显示器发送一个字符
	TEQ	R3,#CR	如果没有进位就返回，继续读取字符
	BNE	READ	

图3-9 读取一行字符并进行显示的ARM程序

### 3.6 子程序

转移与链接（BL）指令通常用于调用一个子程序。它的操作与其他转移指令的操作方式相

同，只是增加了一步。返回地址（BL指令的下一条指令地址）被装入到R14（它实际上作为链接寄存器使用）中。由于子程序可能是嵌套的，所以链接寄存器的内容必须保存在子程序使用的堆栈中。寄存器R13通常作为指向堆栈的指针。

图3-10是对图3-7中的程序采用子程序重写的程序，参数通过寄存器进行传递。调用程序利用寄存器R1和R2将数值列表的大小和第一个数的地址传送给子程序；子程序利用寄存器R0向调用程序返回计算和。该子程序中还用到了寄存器R3。因此，它的内容连同链接寄存器R14的内容一起使用STMFD指令保存到堆栈上。该指令中的后缀FD说明堆栈是向低地址方向增长的，在向堆栈进行压栈之前，堆栈指针R13要预先递减。LDMFD指令恢复寄存器R3中的内容并且将保存的返回地址弹出到PC（R15）中，自动执行返回操作。

调用程序			
	LDR	R1,N	
	LDR	R2,POINTER	
	BL	LISTADD	
	STR	R0,SUM	
	:		
子程序			
LISTADD	STMFD	R13!,{R3,R14}	将R3和返回地址保存到堆栈中的R14中，R13用作堆栈指针
LOOP	MOV	R0,#0	
	LDR	R3,[R2],#4	
	ADD	R0,R0,R3	
	SUBS	R1,R1,#1	
	BGT	LOOP	
	LDMFD	R13!,{R3,R15}	恢复R3并将返回地址载入PC(R15)中

图3-10 采用ARM子程序方式编写图3-7中的程序，通过寄存器传递参数

图3-11a给出的是图3-7中的程序采用堆栈传递参数的子程序重写的程序，调用程序的前四条指令的作用是将参数NUM1和n压入栈中。我们假定NUM1中保存在内存单元POINTER中，子程序中的寄存器R0到R3与在图3-7中的用途相同。它们的内容连同R14中的返回地址一起由子程序中的第一条指令保存到堆栈中。图3-11b给出了在不同时刻栈中的内容。在参数已经压入堆栈并且执行完调用指令（BL）之后，栈顶处于第二级。当子程序中的第一条指令将全部的寄存器保存完之后，栈顶处于第三级。下两条指令利用堆栈中的20和24字节的偏移量将参数装入到寄存器R1和R2中，这时分别从第三级到达n和NUM1级。这时累加和已经在R0中了，由Store指令（STR）将R0插入到栈中，覆盖了NUM1的内容。

122

子程序的最后例子是对嵌套调用情况的处理。图3-12给出了图2-28中程序的ARM代码。第一个和第二个子程序相对应的堆栈结构如图3-13所示，寄存器R12被当作结构指针。为了程序的可读性，在本例中一些寄存器使用了符号名称。寄存器R12（结构指针）、R13（堆栈指针）、R14（链接寄存器）和R15（程序计数器）分别标注为FP、SP、LR和PC。汇编程序指示符RN可以用来定义这些名称。



(假定栈顶在第一级以下)

#### 调用程序

```

LDR    R0, POINTER      NUM1压入堆栈
STR    R0, [R13, # -4]!
LDR    R0, N             n压入堆栈
STR    R0, [R13, # -4]!
BL     LISTADD
LDR    R0, [R13, #4]      把和移动到内存单元SUM中
STR    R0, SUM
ADD    R13, R13, #8      在堆栈上重新移动参数
:

```

#### 子程序

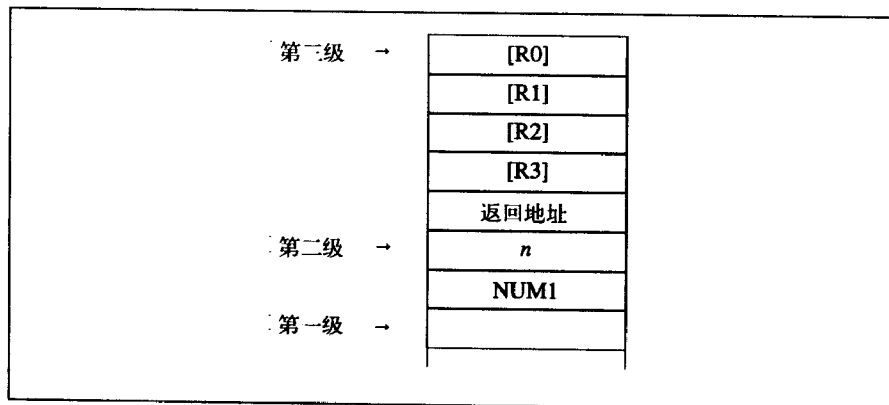
```

LISTADD  STMFD  R13!, (R0-R3, R14)  保存寄存器
        LDR    R1, [R13, #20]      从堆栈中载入参数
        LDR    R2, [R13, #24]
        MOV    R0, #0
LOOP     LDR    R3, [R2], #4
        ADD    R0, R0, R3
        SUBS   R1, R1, #1
        BGT    LOOP
        STR    R0, [R13, #24]      把和放入堆栈
        LDMFD  R13!, (R0-R3, R15)  恢复寄存器并返回

```

123

a) 调用程序和子程序



b) 不同时刻的栈顶

124

图3-11 用ARM子程序编写图3-7中的程序; 采用堆栈传递参数

调用程序和子程序的结构与图2-28相同。ARM的特征如下：两个返回地址和结构指针中的原来内容由每个子程序中的第一条指令保存到栈中，第二个指令将结构指针设置成指向它所保存的值，如图3-13所示。这是与图2-27和图2-29中结构指针相一致的。于是，参数就可以像通常一样在偏移量为8, 12, ...处进行引用了。每个子程序中最后的指令都是恢复结构指针和其他使用过的寄存器的原来内容，以及将从堆栈中弹出的返回地址放到PC中。

内存位置	指令		注释
调用程序			
	:		
2000	LDR	R10,PARAM2	将参数放入堆栈
2004	STR	R10,[SP,#-4]!	
2008	LDR	R10,PARAM1	
2012	STR	R10,[SP,#-4]!	
2016	BL	SUB1	
2020	LDR	R10,[SP]	保存SUB1结果
2024	STR	R10,RESULT	
2028	ADD	SP,SP,#8	在堆栈上重新移动参数
2032	下一指令		
	:		
第一个子程序			
2100	SUB1	STMFD SP!,{R0-R3,FP,LR}	保存寄存器
2104	ADD	FP,SP,#16	载入结构指针
2108	LDR	R0,[FP,#8]	载入参数
2112	LDR	R1,[FP,#12]	
	:		
	LDR	R2,PARAM3	将参数放入堆栈
	STR	R2,[SP,#-4]!	
2160	BL	SUB2	
2164	LDR	R2,[SP],#4	将SUB2的结果弹出放入R2
	:		
	STR	R3,[FP,#8]	将结果放入堆栈
	LDMFD	SP!,{R0-R3,FP,PC}	恢复寄存器并返回
第二个子程序			
3000	SUB2	STMFD SP!,{R0,R1,FP,LR}	保存寄存器
	ADD	FP,SP,#8	载入结构指针
	LDR	R0,[FP,#8]	载入参数
	:		
	STR	R1,[FP,#8]	将结果放入堆栈
	LDMFD	SP!,{R0,R1,FP,PC}	恢复寄存器并返回

图3-12 ARM汇编语言编写的嵌套子程序

3.7 实例程序

在本节中，我们给出第2章描述过的点积、字节排序和链表操作的程序。这些程序都是模仿图2-33、图2-34、图2-37和图2-38中的通用程序所编写的。我们主要描述ARM代码与第2章通用程序中不同部分的特点。

3.7.1 向量点积程序

在图3-14中的前两条指令将A和B向量的地址装入寄存器R1和R2中。它们是3.4.1节中描述的ADR伪指令。如果AVEC和BVEC对于程序来说足够接近，那么就可以根据当前的PC值使用加法指令来生成地址。这里采用相对寻址方式来访问N和DOTPROD的内容，在循环的前两条指令中使用带有写回的传递变址寻址方式。乘法累加指令（MLA）执行必要的算术操作。它将R4和R5

中的向量元素进行相乘，并将结果累加到R0中。

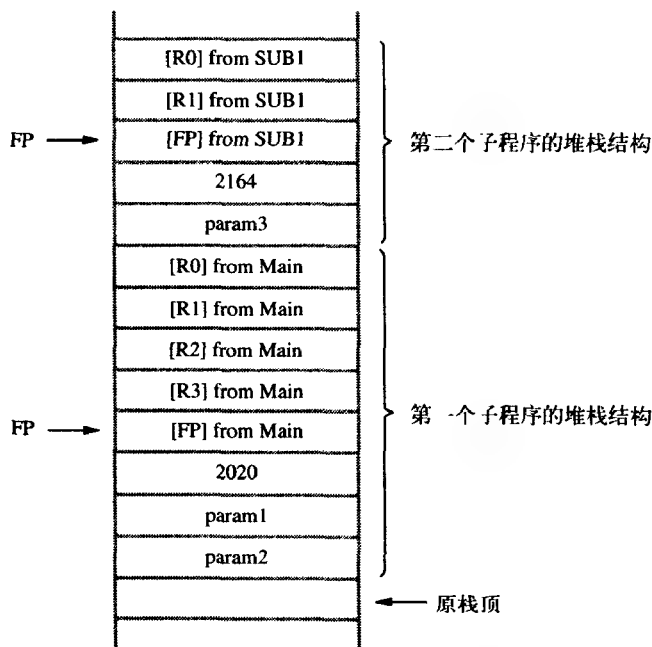


图3-13 图3-12的ARM堆栈结构

LOOP	ADR	R1,AVEC	R1指向向量A
	ADR	R2,BVEC	R2指向向量B
	LDR	R3,N	R3作为循环计数器
	MOV	R0,#0	R0累加点积
	LDR	R4,[R1],#4	载入A元素
	LDR	R5,[R2],#4	载入B元素
	MLA	R0,R4,R5,R0	A、B元素相乘，结果累加到R0中
	SUBS	R3,R3,#1	计数器减1
	BNE	LOOP	如果没完成，转移返回
	STR	R0,DOTPROD	保存点积

图3-14 ARM点积程序

### 3.7.2 字节排序程序

图3-15所示的是字节排序程序。它也使用与图2-34b中程序相同的结构。第一个字节LIST的地址放在寄存器R4，它用于第二到最后一个比较指令中，用来检测内部循环（以  $k$  为变址）何时终止。相应地，R5中的地址是LIST+1，用于最后的一条比较指令来检测外部循环（以  $j$  为变址）何时终止。基址寄存器R2当作变址  $j$  在外部循环的列表末端向回移动的步长，寄存器R3当作变址  $k$  在内部循环每个子列表从末端向回移动的步长。外部循环中将LIST( $j$ )字节装入到寄存器R0，以及内部循环中将LIST( $k$ )字节装入到R1，这两种操作都采用了带有写回的预变址寻址方式。

当LIST( $k$ )必须与LIST( $j$ )进行交换时，ARM指令系统的条件执行特点用于内部循环中是很有优势的。如果LIST( $k$ )比LIST( $j$ )大很多（就像用后缀GT所指示的那样）时，只执行三条指令序列STR、STR和MOV。图2-34b中通用程序到NEXT的前向条件转移，在ARM程序中就不必执行了。

```
for (j = n-1; j > 0; j = j - 1)
{ for (k = j-1; k >= 0; k = k - 1)
  { if (LIST[k] > LIST[j])
    { TEMP = LIST[k];
      LIST[k] = LIST[j];
      LIST[j] = TEMP;
    }
  }
}
```

a) 用C语言编写的排序程序

	ADR	R4,LIST	将列表指针载入到寄存器R4并初始
	LDR	R10,N	化外部循环基址寄存器R2为LIST + n
	ADD	R2,R4,R10	
	ADD	R5,R4,#1	将LIST + 1载入R5
OUTER	LDRB	R0,[R2,# - 1]!	将LIST(j)载入R0
	MOV	R3,R2	初始化内部循环基址寄存器R3为
			LIST + n - 1
INNER	LDRB	R1,[R3,# - 1]!	将LIST(k)载入R1
	CMP	R1,R0	对LIST(k)和LIST(j)作比较
	STRGTB	R1,[R2]	如果LIST(k) > LIST(j), 将LIST(k)
	STRGTB	R0,[R3]	与LIST(j)交换, 并将(新的)LIST(j)
	MOVGT	R0,R1	移到R0中
	CMP	R3,R4	如果k > 0, 重复内部循环
	BNE	INNER	
	CMP	R2,R5	如果j > 1, 重复内部循环
	BNE	OUTER	

b) 用ARM实现的程序

图3-15 ARM字节排序程序

3.7.3 链表的插入和删除子程序

在图3-16和图3-17中的插入和删除子程序与图2-37和图2-38中程序的结构非常相似。在ARM程序中并没有使用普通的前向条件转移。这也是使用具有条件执行指令块（就像图3-15中的字节排序程序一样）的结果。在ARM的两个子程序中都是采用寄存器传递参数的。[127]

这里并没有采用寄存器的通常表示方法来表示，而是采用寄存器助记符的形式来说明它的用途。汇编指示RN就可以这样理解。正如图2-37和图2-38中的程序，RHEAD中包含列表中第一个记录的地址。RNEWREC中包含将要插入的新记录地址。RIDNUM中保存着将要删除记录的地址ID。寄存器RCURRENT和RNEXT中保存着子程序遍历整个列表，找到插入和删除位置的链接地址。[128]

图3-16中的插入子程序是模仿图2-37中子程序的，它具有如下的结构。前三条指令用来在一个空列表中插入新的表头（和表尾）。假定新记录最初的链接字段是0，块中的第三条指令执行的是从子程序向调用程序返回的操作。之后的六条指令判断新记录是否要成为当前链表的新表头，链表是按照ID号递增的方式进行排序。因此，如果当前头记录的第一个字的ID号大于新记录的ID号时，则新记录就作为链表的新表头。如果是这种情况，那么STRGT和MOVGT的条件执

行指令执行相应的链接地址操作。否则，子程序中的其余部分决定新记录应该插入到当前表头之后的什么位置上，其中包含新记录成为表尾的情况。

图3-17给出的是删除子程序。如果要删除的记录是链表的表头，那么前四条指令用来查找表头位置、删除它并返回。否则，子程序的其余部分使用寄存器RCURRENT和RNEXT来遍历链表寻找记录。当发现由RNEXT指向的记录就是要删除的记录时，使用LDREQ/STREQ指令删除该记录。

正如图2-37和图2-38中的普通程序一样，在图3-16中，插入子程序假定新记录的ID号与链表中的任何一个记录的ID号都不相同，图3-17中的删除子程序假定在链表中存在一个ID号与RIDNUM中的内容相同的记录。习题3.23和3.24考虑的是，如果假设不成立，子程序将如何修改才能产生报错信息？

子程序			
INSERTION	CMP	RHEAD,#0	检查列表是否为空
	MOVEQ	RHEAD,RNEWREC	如果为空，插入新记录作为表头
	MOVEQ	PC,R14	
	LDR	R0,[RHEAD]	如果不为空，检查是否新记录成为新表头若是表头则插入
	LDR	R1,[RNEWREC]	
	CMP	R0,R1	
	STRGT	RHEAD,[RNEWREC,#4]	
	MOVGT	RHEAD,RNEWREC	
	MOVGT	PC,R14	
	MOV	RCURRENT,RHEAD	如果新记录处于在当前表头之后，查找其位置
LOOP	LDR	RNEXT,[RCURRENT,#4]	
	CMP	RNEXT,#0	
	STREQ	RNEWREC,[RCURRENT,#4]	新记录成为新表尾
	MOVEQ	PC,R14	
	LDR	R0,[RNEXT]	继续与否
	CMP	R0,R1	
	MOVLT	RCURRENT,RNEXT	继续，重复循环
	BLT	LOOP	
	STR	RNEXT,[RNEWREC,#4]	否则，在当前记录之后插入新记录
	STR	RNEWREC,[RCURRENT,#4]	
	MOV	PC,R14	

图3-16 在链表中插入一条新记录的ARM子程序

子程序			
DELETION	LDR	R0,[RHEAD]	检查要删除的记录是否是表头
	CMP	R0,RIDNUM	
	LDREQ	RHEAD,[RHEAD,#4]	如果是表头，则删除并返回
	MOVEQ	PC,R14	
LOOP	MOV	RCURRENT,RHEAD	否则，继续查找
	LDR	RNEXT,[RCURRENT,#4]	下一记录要删除吗
	LDR	R0,[RNEXT]	
	CMP	R0,RIDNUM	
	LDREQ	R0,[RNEXT,#4]	如果是，删除并返回
	STREQ	R0,[RCURRENT,#4]	
	MOVEQ	PC,R14	
	MOV	RCURRENT,RNEXT	否则，循环返回继续查找
	B	LOOP	

图 3-17 从链表中删除一条记录的ARM子程序

## 部分II 68000实例

在本章的部分II中,我们将通过讨论68000 指令集来描述Motorola 680X0系列处理器的基本体系结构。该系列中包含几款具有不同性能级别的处理器。该系列中所有的成员都具有相同的基本体系结构,但是后期的几款为了提高性能而增添了一些特征。此处使用68000是由于描述上更加简便,以便于可以描述整个系列的显著特征。我们并没有对68000作全面的描述。对于这些信息,读者可以参考生产商的信息<sup>[6]</sup>。我们将集中精力讨论68000的最重要特征,并给出充分的细节让读者能够编写并运行简单的程序。680X0系列中各个成员的不同特征,以及一些为提高性能而引入的特征将在第11章中描述。我们将第2章中的程序用68000汇编语言进行编写来说明68000体系结构的各种特点。

[130]

### 3.8 寄存器与寻址方式

68000处理器的特点是,由于处理器芯片中有16条与内存相连的数据引脚,所以它有一个16位的外部字长。但是数据的处理是在处理器的32位字长的寄存器中进行的。该系列中的更先进型号有68020、68030和68040,这些处理器中容入了更大的芯片包,并具有32个外部数据引脚。这样,它们可以处理的内部和外部数据都是32位的。Tatak<sup>[7]</sup>对680X0系列的成员作了描述,并着重强调了68040这款处理器。

#### 3.8.1 68000寄存器结构

如图3-18所示,68000寄存器结构具有8个数据寄存器和8个地址寄存器,长度都是32位。数据寄存器可以作为通用累加器和计数器来使用。

68000指令可以处理三种不同长度的操作数。其中32位的操作数占一个“长字”,16位操作数占一个“字”,8位操作数占一个“字节”。当一条指令使用的操作数是字节或字时,操作数要放在寄存器的低位部分。在大多数情况下,这种指令不会影响寄存器中的高位部分,但是,有些指令将较短的操作数的符号位扩展到高位部分。

地址寄存器中保存着用于确定内存操作数的信息。该信息可以用一个长字或一个字来指定。当一个给定内存位置的地址处于地址寄存器中时,寄存器就用作一个指针来指向这个位置。此外,地址和数据寄存器都可以当作变址寄存器。地址寄存器A7具有特殊的功能,它是处理器堆栈的指针。该寄存器的作用我们在3.13节中讨论。

地址寄存器和地址的计算都是32位的。但是在68000中,地址中最低位开始的24位用作外部对内存的访问。68020、68030和68040处理器具有32位的外部地址线和32位的数据线。

图3-18中所示的最后一个寄存器是处理器状态寄存器SR。该寄存器中有五个条件码位(在3.11.1节中描述)、三个中断位(在第4章描述)和两个模式选择位(在3.13节中描述)。

#### 3.8.2 寻址方式

68000计算机的内存是按照16位的字来组织的,同时是字节可寻址的。两个连续的字可以看作是一个单个的32位长字。内存中的地址分配如图3-19所示。一个字必须从偶地址开始,也就是它的地址必须是偶数,所采用的是big-endian地址分配方式。一个字中的高位字节具有和字相同的地址,但是,低位字节的地址是接下来的较高地址。

[131]

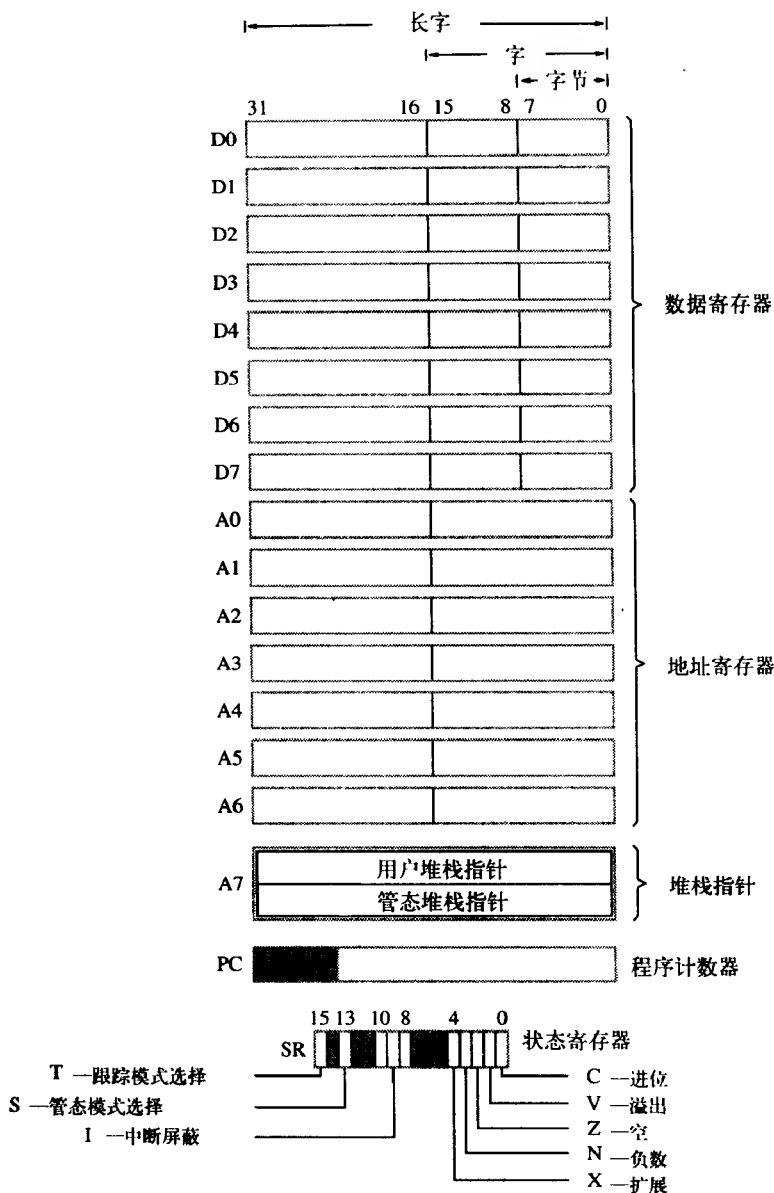


图3-18 68000寄存器结构

由于68000生成的是24位地址，所以它的寻址空间为 $2^{24}$  (16 777 216或16M) 字节。这个寻址空间可以看成是由512 ( $2^9$ ) 个页 (每页有32K ( $2^{15}$ ) 字节) 构成。于是，十六进制地址0到7FFF构成第0页，8000到FFFF构成第2页，依次类推。最后一个页由地址FF8000到FFFFFF构成。

68000中包含几种寻址方式，其中包括在2.5节中讨论过的内容。很多68000指令适合使用16位的字，但是有些还需要额外的字来保存额外的寻址信息。指令中的第一个字是操作码，它指定了要执行的操作以及一些寻址信息。其余的寻址信息 (如果有) 在后续的字中给出。可用的寻址方式定义如下：

**立即方式**——操作数包含在指令中，可以指定四种长度的操作数。字节、字和长字是紧跟在操作码之后的。第四种长度的操作数是由非常小的数构成，它可以直接包含在一些指令的操作码中。

**绝对方式**——操作数的绝对地址是在指令中紧跟在操作码之后的。该方式有两种版本——长方式和短方式。在长方式中，24位地址是明确指定的。在短方式中，指令给出一个16位的值作为地址的低16位。将该值的符号位进行扩展来提供地址的高8位。由于符号位是0或1，所以在短方式中可以访问的地址空间只有两个页。它们是第0页和第FF8页，每个容量是32K字节。

**寄存器方式**——操作数在指令指定的处理器寄存器中。

**寄存器间接方式**——操作数的有效地址在指令指定的地址寄存器中。

**自动递增方式**——操作数的有效地址放在指令指定的地址寄存器 $An$ 中。在操作数访问结束之后， $An$ 中的内容将增加1、2或4，这要视使用的操作数是字节、字还是长字而定。

**自动递减方式**——指令中指定的地址寄存器 $An$ 的内容根据使用的操作数是字节、字或长字，而减去1、2或4。操作数的有效地址是 $An$ 被减值后的内容。

**基址变址方式**——在指令中指定一个16位的带符号的偏移量和地址寄存器 $An$ 。偏移量与 $An$ 内容的相加和就是操作数的有效地址。

**完全变址方式**——指令中给出一个带符号的8位偏移量，一个地址寄存器 $An$ 和一个变址寄存器 $Rk$ （可以是地址或数据寄存器）。操作数的有效地址是偏移量与寄存器 $An$ 及 $Rk$ 内容相加的和。在原地址中可以使用完整的32位，或 $Rk$ 的低16位符号扩展方式的地址。

**基址相对方式**——该方式与基址变址方式相同，只是它使用的是程序计数器，而不是地址寄存器 $An$ 。

**完全相对方式**——该方式与完全变址方式相同，只是它使用的是程序计数器，而不是地址寄存器 $An$ 。

寻址方式及其汇编语法总结在表3-2中给出。

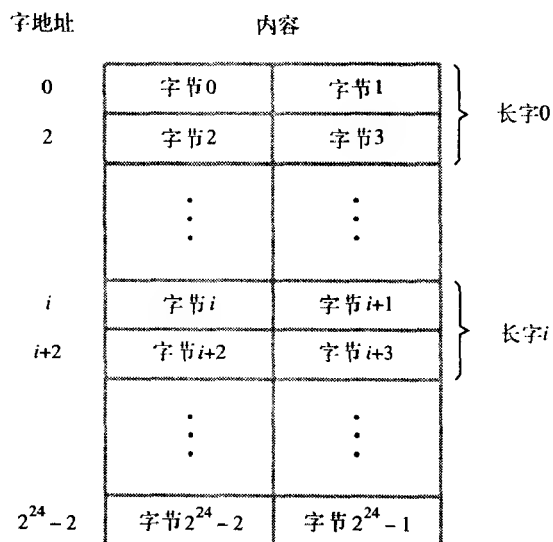


图3-19 68000中可寻址单元图

133

表3-2 68000寻址方式

名 称	汇 编 语 法	寻 址 功 能
立即	#Value	Operand = Value
绝对短	Value	EA = 符号扩展的 WValue
绝对长	Value	EA = Value
寄存器	Rn	EA = Rn 即 Operand = [Rn]
寄存器直接	(An)	EA = [An]
递增	(An)+	EA = [An] 递增An
递减	-(An)	递减An EA = [An]



(续)

名 称	汇 编 语 法	寻 址 功 能
变址基址	WValue( <i>An</i> )	$EA = WValue + [An]$
变址完全	BValue( <i>An</i> , <i>Rk</i> . <i>S</i> )	$EA = BValue + [An] + [Rk]$
相对基址	WValue( <i>PC</i> ) 或Label	$EA = WValue + [PC]$
相对完全	BValue( <i>PC</i> , <i>Rk</i> . <i>S</i> ) 或Label( <i>Rk</i> )	$EA = BValue + [PC] + [Rk]$

EA = 有效地址

Value = 明确给出或用标号表示的数

BValue = 8位

WValue = 16位

*An* = 地址寄存器*Rn* = 地址或数据寄存器*S* = 长度指示符: W表示符号扩展的16位字, L表示32位长字

注意有两个版本的变址方式, 基址变址方式与图2-13描述的方式相对应。完全变址方式包括指令中的两个寄存器的内容和一个偏移常量。偏移常量的长度在基址方式中是16位, 在完全方式中是8位。

在完全变址方式中, 第二个寄存器*Rk*有两种使用方式: 使用全部32位或只使用其中的低16位。通过在寄存器名称后面加上一个长度指示符来指示汇编程序所使用的是何种方式——L代表长字, W代表字, 例如, D1.L或D1.W。如果不给出提示, 默认的方式就是后一种情况。在计算一个32位有效地址时所使用的是一个16位的字, 要将这个字进行符号扩展。

在两种变址方式中, 都可以使用程序计数器来代替地址寄存器。寻址方式称为相对寻址方式, 因为有效地址是以操作数与涉及到该操作数的指令之间的距离形式指定的。考虑指令

134

ADD 100 ( PC, A1 ), D0

当编码成机器指令形式时, 该指令由两个字构成。操作码说明这是一条加法指令, 目标寄存器是数据寄存器D0, 原操作数使用的是完全相对寻址方式。第二个字也称为扩展字 (extension word), 说明寄存器A1用作变址寄存器, 其中包含8位的偏移量值100。

假定前面的指令保存在1000处, 寄存器A1包含的值是6, 如图3-20所示。当该指令的操作码已经取出并由处理器对其进行编码的时候, 程序计数器指向扩展字, 这就意味着程序计数器中的值是1002。因此, 原操作数的有效地址是:

$$\begin{aligned}
 EA &= [PC] + [A1] + 100 \\
 &= 1002 + 6 + 100 \\
 &= 1108
 \end{aligned}$$

135

图3-20指出了使用这种寻址方式是如何访问数组中的一个数据项的。偏移量说明了在数组中的第一项与指令的距离。之后, 变址寄存器给出了那一点到所需操作数 (在数组中的第四个字) 的距离。

我们使用了明确的格式给出了相对寻址方式。大多数的汇编程序允许使用更加简单的形式来指定这种寻址方式。首先, 必须通过一个适当的汇编指示提示汇编程序, 在程序中某个给定的部分中使用了相对寻址方式。之后, 在名称ARRAY已经分配了一个值1102以后, 图3-20中的指令就可以写成

## ADD ARRAY (A1), D0

汇编程序将该源操作数解释为使用完全相对寻址方式，并按照图中指示的那样计算偏移量。汇编程序不知道（也无需知道）在执行指令时寄存器A1中的内容。例如该指令可以用在一个程序循环中，在这种情况下A1可以用来访问数组中的连续元素。

完全相对寻址方式中，由于偏移量采用8位补码值来表示，因此它的值限定在-128到+127之间。

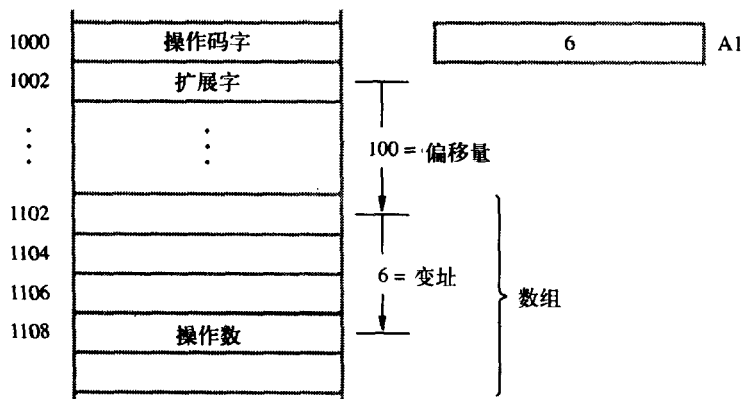


图3-20 对于指令ADD 100 (PC, A1), D0, 68000完全相对寻址方式的例子

### 3.9 指令

68000 ISA提供了一个广泛的指令系统，其中的大部分指令可以操作三种不同长度的操作数。附录C给出了指令系统的总结。大部分的指令可以按照统一的方式使用所有的寻址方式。指令的这一特征称为正文。

136

68000具有单操作数和双操作数指令。一个双操作数指令写为：

OP src, dst

该指令使用源操作数和目标操作数来执行操作OP，结果保存在目标单元中。图3-21给出了一个例子，其指令是：

ADD #9, D3

该指令完成的动作是：

$dst \leftarrow [src] + [dst]$

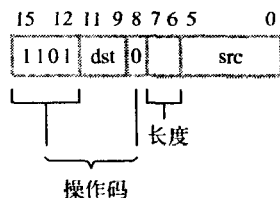
137

它的结果是将值9与寄存器D3的内容相加并将结果保存到D3中。

图3-21a描述的是ADD指令的一般格式。其中的源操作数或目标操作数中的一个必须在数据寄存器Dn中。第二个操作数可以在寄存器中，也可以在一个内存单元中。在表C-4中给出了允许的组说明。由于两个操作数中至少有一个使用了8个数据寄存器中的一个，所以3位标识字段就足够用了。其他的操作数根据表C-1中的说明来指定。在我们的例子中，目的寄存器D3用9到11位的二进制形式表示成011；源操作数是立即数，用第0到第5位的111100的形式来标识。

所需操作数的长度使用2位字段来标明。在例子中，操作数的长度没有在汇编语言语句中明确标明，这种情况下汇编程序假设的默认值是16位字。根据表C-3，字长度的操作数以01形式来指示。

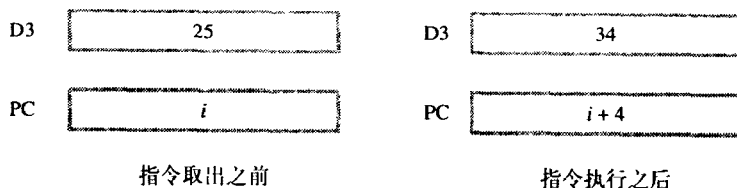
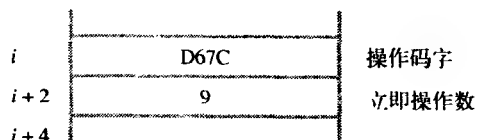
通过上面的讨论, ADD指令的操作码就是1101011001111100, 用十六进制表示就是D67C, 如图3-21b所示。



a) ADD指令的操作码字的格式

二进制 1101011001111100  
十六进制 D 6 7 C

b) 操作码字的编码



c) 指令执行结果

图3-21 68000指令 ADD #9, D3

源操作数是立即数9, 紧跟在操作码之后给出, 如图3-21c所示。在取指令之前, 程序计数器指向操作码的地址*i*。在从内存中每次取出一个字时, PC的内容增加2。这样, 当指令执行完成时, PC指向下一条指令的操作码的地址*i+4*。

采用相同格式的另一条类似指令是减法指令SUB, 执行的操作

$$\text{dst} \leftarrow [\text{dst}] - [\text{src}]$$

正如表C-4所示, ADD和SUB指令可以对两个操作数中的一个进行非常灵活的指定。但是, 第二个操作数必须放在数据寄存器中。大多数其他的双操作数指令具有同样类型的限制。只有一条指令, 它的两个操作数可以采用大多数的寻址方式来指定, 该指令就是传送指令MOVE, 它执行的动作是

$$\text{dst} \leftarrow [\text{src}]$$

现在考虑一个简单的  $C \leftarrow [A] + [B]$  的任务, 如图2-8所示。要求的任务可以按照如下方式执行

MOVE A, D0

ADD B, D0

MOVE D0, C

这些指令可能存储在68000计算机的内存中，如图3-22所示。图中将地址和操作数的值用十六进制来表示。假定操作数长度是16位，它们的地址采用绝对寻址方式来指定。注意，由于所需的地址不能表示成16位的形式，所以应该使用绝对寻址方式中的长寻址方式。根据图3-19中的约定，将32位地址中的高16位放到低地址字中，低16位放到高地址字中。

3.10 汇编语言

在2.6节中讨论的汇编语言通常也适用于68000汇编语言。这里主要阐述一些细微的不同之处和额外的部分。

由于68000指令可以处理三种不同长度的操作数，所以汇编指令必须要指定所需的长度。通过在操作助记符的后面加上长度指示符来实现这种功能。长度指示符中的L表示长字，W表示字，B表示字节。因此，如果加法指令使用的操作数是一个长字，其操作助记符就写作ADD.L。如果不给出长度指示，那么操作数的长度就是一个字。这就意味着指令ADD.W #20, D1与指令ADD #20, D1是相同的。

除非使用十六进制前缀\$或二进制前缀%，否则程序中的数值都假定使用十进制来表示。汇编程序将单引号之间的字母数字字符用它们对应的ASCII码来替换。引号之间的一个字符串中可以有多个字符。例如，字符串‘STRING3’就是一个有效字符串。

在2.6节中讨论的全部汇编指示除了在名称上有少许的不同之外都可以使用。指令或数据块的起始地址采用ORG指示符来指定。EQU指示符相当于数值的名称。采用DC (Define Constant) 指示符将数据常量来插入到一个目标程序中。指示符后面所跟的长度标识是专门用来说明数据项的长度，一个指示符可以定义多个数据项。例如指示符

```
ORG 100
PLACE DC.B 23, $4F, %10110101
```

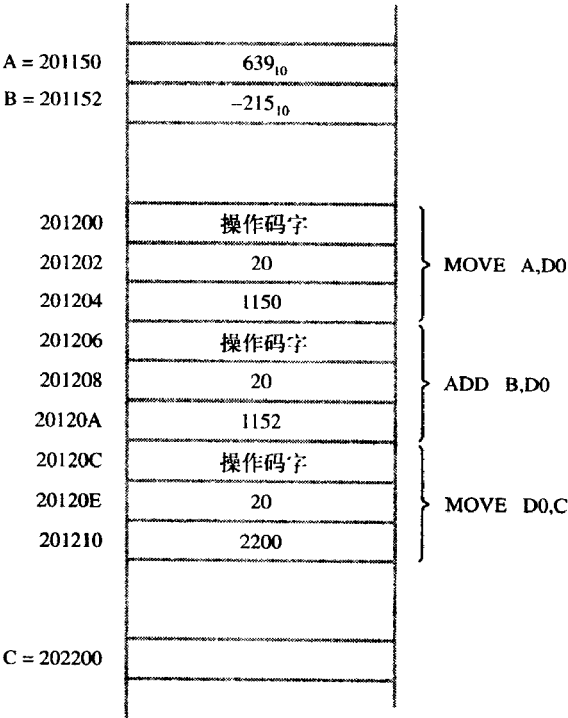
结果是将十六进制值17(23<sub>10</sub>)、4F和B5分别装入到内存中的100、101和102中。标号PLACE被分配的值是100。

内存块可以通过DS (Define Storage) 指示符来预留。例如指示符

```
ARRAY DS.L 200
```

将200个长字的空间预留出来，同时将第一个长字的地址和名称ARRAY关联起来。

在图3-23中给出的是对应于图3-22 的一个68000汇编语言程序的简单例子。



执行之后，[202200] = 424<sub>16</sub>

图3-22 C ← [ A ] + [ B ]的68000程序

	内存地址 标号	操作	地址或数 据信息
汇编指示	C	EQU	\$202200
		ORG	\$201150
	A	DC.W	639
	B	DC.W	-215
		ORG	\$201200
生成机器指令的语句		MOVE	A,D0
		ADD	B,D0
		MOVE	D0,C
汇编指示		END	

图3-23 用68000汇编语言表示的图3-22中的程序

140

### 3.11 程序流控制

转移指令是用来实现程序结构的，如if语句与循环。通常，转移指令要测试一个转移条件，并根据测试结果转向一条或两条可能的路径继续执行。测试的条件与刚刚执行的操作结果有关。

#### 3.11.1 条件码标志

68000中有五个存储在状态寄存器中的条件码标志，如图3-18所示。68000中除了有在2.4.6节中描述的N、Z、V和C标志以外，还有第五个标志X（扩展）。它的设置方式与C标志相同，但是它不受任何指令的影响。当我们进行高精度操作时，这种表面上的复制确实非常方便，这些内容我们将在第6章中介绍。

附录C中的表C-4给出了每条指令会影响哪些标志。如果有进位时将C和X标志设置成1，它们是加法操作执行结果中最重要的位。在执行一条减法指令中没有进位时，将C和X标志设置成1，代表借位信号。由于操作数可以说明成三种长度中的任何一种，所以这两个标志位要分别依赖于字节、字或是长字对应的来自于第7、15和31位上的进位。MOVE指令根据传送的操作数来设置N和Z标志，并清除C和V标志。除非指定的目标是状态寄存器本身，否则MOVE指令不会影响X标志位。

#### 3.11.2 转移指令

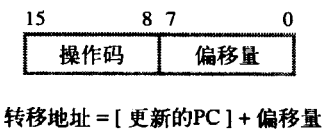
在条件转移指令中，如果条件满足，那么条件转移指令就继续执行转移目标地址处的指令。这个转移目标地址由操作数字段中的转移偏移量决定。否则，如果转移条件不满足，那么将执行紧跟在转移指令之后的指令。68000给出了两类偏移量的转移指令。第一类是包含在操作码字中8位的短偏移量。这些指令用于计算转移地址时程序计数器中的值是+127或-128以内的转移目标地址。回想一下，从内存中每次取出一个字时，PC的内容就会增加，这就意味着偏移量定义的是紧跟在转移指令操作码之后的字之间的距离。第二类是16位的偏移量，紧跟在操作码之后的字中。该类型为转移目标的定位提供了更大的范围（ $\pm 32K$ ）。此时，偏移量是扩展字到转移目标之间的距离。

141

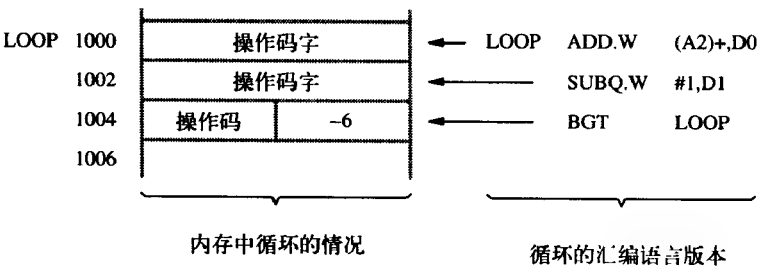
图3-24 说明的是短偏移量转移指令的用法。图中给出了图2-16的程序循环是如何在68000处

理器上实现的。注意图2-16中的程序使用的是减1指令。由于68000中没有这种指令，我们使用的是快速减法指令SUBQ，该指令是从寄存器D1中减去立即数1。一个3位的立即操作数包含在SUBQ指令的操作码中；这样只需一个字就可以表示这条指令。

68000有16个条件转移指令，每条指令都可以带8位和16位的偏移量。同时还有一个非条件转移指令BRA，它的转移总是满足的。表C-5和表C-6中给出了这些指令的详细内容。



a) 短偏移量转移指令的格式



当计算转移地址时，[ PC ] = 1006  
转移地址 = 1006 - 6 = 1000

b) 在图2-16的循环中使用转移指令的例子

图3-24 68000 短偏移量转移指令

图3-25给出了采用68000编写的图2-16中的加法程序。程序中分别使用数据寄存器D0和D1作为累加器与计数器，使用地址寄存器A2指向从内存中取出来的操作数。注意由于在自动递增寻址方式中只允许使用地址寄存器，所以这里使用了地址寄存器。

	MOVE.L	N,D1	N包含n，n是要相加的项的个数，D1用作决定循环执行次数的计数器
	MOVEA.L	#NUM1,A2	A2用作指向表项的指针。其初始化时指向NUM1，第一个项的地址
LOOP	CLR.L	D0	D0用作累计相加和
	ADD.W	(A2)+,D0	与D0相加的连续的数值
	SUBQ.L	#1,D1	计数器减1
	BGT	LOOP	如果[D1] ≠ 0，再次执行循环
	MOVE.L	D0,SUM	把和保存到SUM中

图3-25 采用68000编写的图2-16中的加法程序

递减和转移指令

68000中除了一些通常的转移指令以外，还有一套更加复杂的包含着计算机制的指令。这些技巧对于实现循环控制是非常有用的。这些指令使用下面的格式

DBcc Dn, LABEL

其中的后缀表示转移条件。例如，如果采用GT来代替cc，那么只有是“大于”指令时，指令DBGT将递减并转移。在表C-6中给出了全部可能转移指令。这些指令中的条件转移使用方式与其他转移指令中的使用方式相反。其行为如下：

如果cc指定的条件满足，那么立即执行在DBcc指令之后的指令。

如果cc指定的条件不满足，寄存器Dn中最低16位减1。如果结果等于-1，下一步就执行DBcc指令之后的指令。如果结果不等于-1，转移指令转向在LABEL处的指令。

由于DBcc指令决定是否转移取决于两个条件而不是一个条件，所以它的功能比普通的转移指令更加强大。如果使用普通的转移指令来完成同样的动作，必须使用三条指令的序列来完成：首先，用一个转移指令来检测cc条件；然后，使用一个将计数器内容减1的指令；最后，再用一条转移指令根据减1操作的结果来决定是否转移。例如，指令

143

DBcc D3, LOOP

下一指令

等价于序列

Bcc NEXT

SUBQ #1, D3

BGE LOOP

NEXT 下一指令

考虑DBcc指令一种非常有用的方式是将其看成提供循环控制的简便手段，当给定的条件满足时可以较早地从循环中退出。循环执行的次数是由计数器寄存器中的内容限制的，在前面给出的例子中就是D3。

一个DBcc指令DBF（如果条件是FALSE就递减并转移），使用的测试条件总是FALSE。这样，决定是否执行转移仅仅基于计数器寄存器的内容递减之后的结果。一个循环总是执行预先定义的次数是非常有用的。它还有第二个名字DBRA（递减并总是转移）。

为了说明递减和转移指令的作用，我们将图3-25中的程序采用DBRA指令重新编写，如图3-26所示。寄存器D1初始化为图3-25中的值n。但是，由于当计数器寄存器中的内容等于或大于0时，DBRA指令才执行转移，所以在图3-26中的寄存器D1被初始化成n-1。两个程序中具有相同的指令总数，但图3-26中的程序由于使用的循环更短，所以使得程序的执行时间也更短。

	MOVE.L	N, D1	将n-1放入计数器寄存器D1
	SUBQ.L	#1, D1	
	MOVEA.L	#NUM1, A2	
	CLR.L	D0	
LOOP	ADD.W	(A2)+, D0	
	DBRA	D1, LOOP	循环直至[D1] = -1
	MOVE.L	D0, SUM	

144

图3-26 图3-25中程序的另一个可选择的68000程序

### 3.12 I/O操作

68000处理器所需的I/O设备接口中的全部状态和数据缓冲区，都是像内存一样是可寻址的。

这就意味着68000计算机中的程序控制I/O可以按照2.7节中一般的讨论内容来实现。

假设键盘状态寄存器INSTATUS的b<sub>3</sub>位包含输入控制标志SIN。从键盘的输入操作通过下面的指令序列来实现

```
READWAIT BTST.W #3, INSTATUS
          BEQ    READWAIT
          MOVE.B DATAIN, D1
```

按位检测指令BTST检测目标操作数中一位的状态，并设置条件码标志Z。在本例中，被检测位的位置是在第一个操作数中指定的。

假设显示状态寄存器OUTSTATUS中的b<sub>3</sub>位包含输出控制标志SOUT，可以通过下面的指令序列将寄存器D1中的字符发送到显示器：

```
WRITEWAIT BTST.W #3, OUTSTATUS
          BEQ    WRITEWAIT
          MOVE.B D1, DATAOUT
```

一个从键盘读取一行字符并将它们保存到内存，然后将它们在显示器上显示的68000程序如图3-27所示。这个程序是模仿图2-20中的程序编写的。程序中假定遇到回车时一行结束。字符保存在内存中以LOC为起始地址的字节处。

	MOVEA.L	#LOC,A1	初始化指针寄存器A1，使其保存字符在内存中存储位置的起始地址
READ	BTST.W	#3, INSTATUS	等待一个字符输入到键盘缓冲DATAIN中
	BEQ	READ	
	MOVE.B	DATAIN,(A1)	把DATAIN中的字符转移到内存中（这将清SIN为0）
ECHO	BTST.W	#3,OUTSTATUS	等待显示器就绪
	BEQ	ECHO	
	MOVE.B	(A1),DATAOUT	将刚刚读入的字符移到输出缓冲寄存器中（这将清SOUT为0）
	CMPI.B	#CR,(A1)+	检查刚读入的字符是否为CR（回车），如果不是CR，转移返回，读取下一个字符。
	BNE	READ	同时增加指针以指向下一个字符

图3-27 读取一行字符并进行显示的68000程序

3.13 堆栈和子程序

堆栈可以像2.8节中说明的那样使用任何一个地址寄存器作为指针来实现。自动递增与自动递减寻址方式使得这个过程更加简便。一个特殊的寄存器A7被设计成为处理器堆栈指针，并且该处理器指向的堆栈称为处理器堆栈。就是这个栈用在了处理器自动执行的所有堆栈操作中，例如在子程序链接情况下。

图3-18中给出的两种不同的32位寄存器A7。68000提供了两种不同的操作模式，用户模式和管态模式。每种模式中都有自己的处理器堆栈指针A7。在管态模式中，处理器可以执行全部的机器指令。在用户模式中不可以执行那些特权指令。应用程序通常在用户模式下运行，而系统软件则使用管态模式。处理器状态寄存器中的S位决定两种模式中的哪一种处于活动状态，也就



是说,当前使用的是两个A7寄存器中的哪一个。

转移到子程序(Branch-to-Subroutine, BSR)指令用于子程序的调用。它的实现方式与其他转移指令相同,但也会将程序计数器的内容压入栈中。它的转移目的地址是子程序中的第一条指令。当子程序结束时,使用一条从子程序返回指令(RTS)返回到调用程序中,将栈顶的返回地址弹出到程序计数器中。BSR和RTS指令允许子程序链接机制得以实现,这部分在2.9节中有一般性的描述。

图3-28给出了如何使用子程序编写图3-26中的程序,并通过寄存器来传送参数。列表地址与表中的项数是通过寄存器A2和D1传送给子程序的。在执行完加法之后,子程序返回寄存器D0中的和。

图3-29给出了如何使用子程序编写图3-26中的程序,通过寄存器A7指向的堆栈来传送参数。MOVEM(传送多个寄存器)指令保存和恢复寄存器A2、D1和D0。这些寄存器在堆栈中的保存顺序,如图3-29b所示。第一个MOVEM指令使用自动递减寻址方式,将指定的寄存器压入栈中。第二个MOVEM指令使用自动递增寻址方式将堆栈中保存的值弹出,并将它们按照相反的顺序保存到寄存器中。

现在考虑子程序嵌套的情况,其中一个子程序调用另一个子程序,如图2-28中所示。图3-30给出了这个例子的68000代码,子程序SUB1和SUB2的堆栈结构如图3-31所示。主程序调用子程序SUB1。在调用指令BSR执行之前,主程序将SUB1要使用的参数param2和lparam1压入栈中。

146

调用程序			
	MOVEA.L	#NUM1,A2	将地址NUM1放到A2中
	MOVE.L	N,D1	将项数 $n$ 放到D1中
	BSR	LISTADD	调用子程序LISTADD
	MOVE.L	D0,SUM	将和存到SUM中
	下一指令		
	:		
子程序			
LISTADD	SUBQ.L	#1,D1	调整计数为 $n-1$
	CLR.L	D0	
LOOP	ADD.W	(A2)+,D0	在D0中累积和
	DBRA	D1,LOOP	
	RTS		

图3-28 将图3-26中的程序编写成一个68000子程序;通过寄存器传递参数

子程序通过创建自己的堆栈结构开始。特定指令

LINK Ai, # disp

将寄存器Ai设置为结构指针,并执行如下操作:

1. 将寄存器Ai的内容压入处理器栈中。
2. 将处理器堆栈指针A7的内容复制到寄存器Ai中。
3. 将指定的位移量加到寄存器A7中。

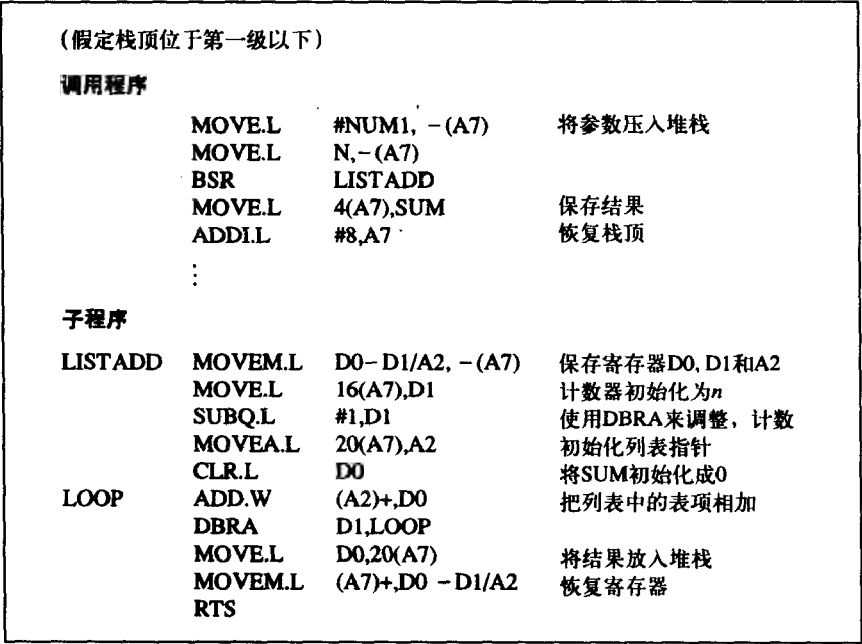
如果位移量是一个负数,将会导致栈顶向上移动(低地址方向),这样,就创建了子程序中

局部变量使用的空堆栈空间。这些变量可以采用结构指针寄存器A*i*的变址寻址方式来访问。在子程序的最后，UNLK（断开连接）指令与LINK指令的动作相反。它把A*i*装到A7中，这样，降低栈顶回到与偏移量相加之前的位置，然后将寄存器A*i*的内容从堆栈中弹出并放回到A*i*中。

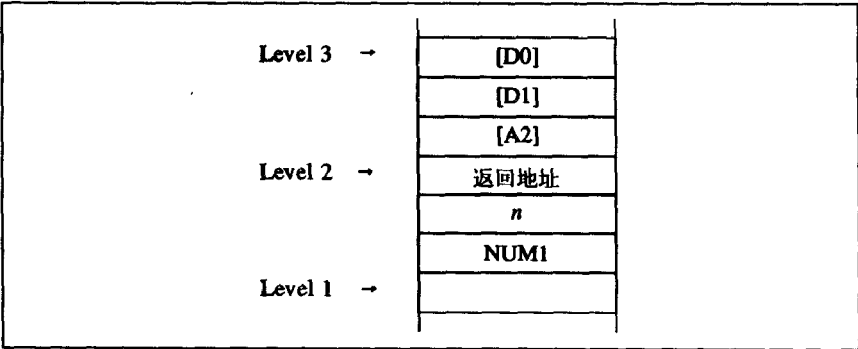
在图3-30的例子中，我们假定子程序只使用寄存器就可以执行它们的任务，所以不需要堆栈上的工作空间。因此，每个子程序的开始都是指令

LINK A6, #0

147



a) 调用程序和子程序



b) 不同时刻的堆栈内容

图3-29 将图3-26中的程序编写成一个68000子程序；通过堆栈传递参数

148

定义寄存器A6作为堆栈结构指针，将A7指向A6中原来保存内容的位置。这条指令与图2-28中子程序开始处的两个传送指令执行的操作相同。在每一种情况下，它都是跟在MOVEM指令的后面，将子程序所需寄存器的内容保存到堆栈中。

图3-30中程序的其余部分都是采用68000指令按照图2-28中的程序直接实现的。在图3-30中的程序执行时，执行的结果项数保存到堆栈上，如图3-31所示。主程序将两个参数压入堆栈中，之后，BSR指令将返回地址2014压入到堆栈中。注意由于到SUB1的偏移量很小，可以用8位来表示，所以BSR指令可以放在2012的一个字中。SUB1中的LINK和MOVEM指令保存结构指针A6和四个其他寄存器的内容。

内存位置	指令	注释
调用程序		
2000	MOVE.L	PARAM2, -(A7)
2006	MOVE.L	PARAM1, -(A7)
2012	BSR	SUB1
2014	MOVE.L	(A7), RESULT
2020	ADDI.L	#8, A7
2024	下一指令	恢复堆栈级别
...		
第一个子程序		
2100 SUB1	LINK	A6, #0
2104	MOVEM.L	D0-D2/A0, -(A7)
	MOVEA.L	8(A6), A0
	MOVE.L	12(A6), D0
	...	
	MOVE.L	PARAM3, -(A7)
2160	BSR	SUB2
2164	MOVE.L	(A7)+, D1
	...	
	MOVE.L	D2, 8(A6)
	MOVEM.L	(A7)+, D0-D2/A0
	UNLK	A6
	RTS	返回
第二个子程序		
3000 SUB2	LINK	A6, #0
	MOVEM.L	D0-D1, -(A7)
	MOVE.L	8(A6), D0
	...	
	MOVE.L	D1, 8(A6)
	MOVEM.L	(A7)+, D0-D1
	UNLK	A6
	RTS	返回

图3-30 用68000汇编语言编写的嵌套子程序

在子程序SUB1调用SUB2之前，它将一个参数param3压入到堆栈中，返回地址2164由BSR指令压入到栈中。由于到目标地址的偏移量超过了8位所能表示的范围，所以，BSR指令占用了两个字。当每个子程序完成自己的任务时，就恢复保存的寄存器内容并返回。在控制返回到主程序以后，由SUB1（覆盖param1）放到堆栈中的结果将保存到内存中的RESULT处。之后，通过ADDI.L指令，堆栈指针A7将恢复它的原来值并指向图3-31中的原栈顶元素。

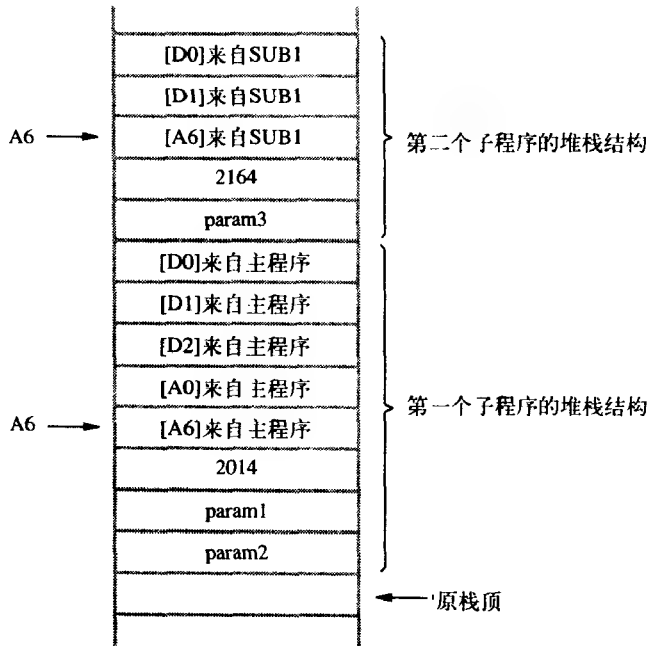


图3-31 图3-30的68000堆栈结构

150

### 3.14 逻辑指令

在上节中，我们讨论了操作数传送指令以及加法和减法算术操作指令。这些指令中的操作数都具有32、16或8位的固定长度。在有些应用中需要处理其他长度的数据，也许是个别的位，对这些数据执行逻辑操作。68000中有用于此目标的几条指令。特别在68000中有可以执行逻辑与、或、异或操作的指令，以及具有几种不同方式的移位和循环移位操作的指令。

为了说明逻辑指令的用法，我们来看两个例子。假定寄存器D1中保存一些32位的模式，我们想要判断该值中的 $b_{18}$ 到 $b_{14}$ 位是否是11001。可以采用指令

```
AND.L  #$7C000, D1
CMPL.L  #$64000, D1
BEQ     YES
```

第一条指令是对原操作数和目标操作数个个别位进行逻辑与的操作，并将结果保存到寄存器D1中。十六进制数7C000的 $b_{18}$ 到 $b_{14}$ 位是1，其他的位是0。这样与操作的结果是寄存器D1中 $b_{18}$ 到 $b_{14}$ 的五位保持原值不变，其余的位全变成0。后面的比较指令是用来检测这五位是否与所期望模式对应的。

#### 数字打包程序

作为第二个例子，再次考虑图2-31中的BCD数字打包程序。该程序的68000代码如图3-32所示。在寄存器D0和D1中分别装有两个ASCII码字节。LSL指令将D0中的字节左移四位，低位部分用0填充。在该指令操作数字段中的第一项是操作数要传送的位数。表C-4给出的数也可以在其他的数据寄存器中指定。因此，如果D2中的内容已经预先设置成4的话，下面的指令也可以达到同样的效果

```
LSL.B  D2, D0
```

ANDI指令将第二个字节的高四位设置成0。最后，所期望的4位BCD码用“或”指令合并到D1中，

151

并保存到内存的PACKED中。

MOVEA.L	#LOC,A0	A0指向数据
MOVE.B	(A0)+,D0	将第一个字节载入D0
LSL.B	#4,D0	左移4位
MOVE.B	(A0),D1	将第二个字节载入D1
ANDI.B	#\$F,D1	将高4位清0
OR.B	D0,D1	将两个数连接
MOVE.B	D1,PACKED	保存结果

图3-32 打包BCD数的68000逻辑指令的使用

### 3.15 实例程序

在本节中，我们给出在第2章中描述的点积、字节排序和链表操作的68000版本的程序。

#### 3.15.1 向量点积程序

图2-33给出的是计算两个向量AVEC和BVEC的点积程序。它的68000版本如图3-33所示。两个程序中除了控制循环的DBRA指令不同以外，其余的基本相同。为了使计数器D0的内容减1，所以必须使用这条指令。这正如在3.11.2节中描述的那样。

注意MULS指令将两个带符号的16位数相乘，产生一个32位的乘积。我们假定向量元素用16位的字表示，点积也是16位的。所有地址都按照32位来处理。

	MOVEA.L	#AVEC,A1	第一个向量地址
	MOVEA.L	#BVEC,A2	第二个向量地址
	MOVE	N,D0	元素个数
	SUBQ	#1,D0	调整计数为DBRA
	CLR	D1	将D1用作累加器
LOOP	MOVE	(A1)+,D2	从向量A中提取元素
	MULS	(A2)+,D2	与向量B中元素相乘
	ADD	D2,D1	将乘积累加
	DBRA	D0,LOOP	
	MOVE	D1,DOTPROD	

图3-33 计算两个向量点积的68000程序

152

#### 3.15.2 字节排序程序

现在我们考虑图2-34中给出的字节排序的程序。该程序采用直接选择算法将表中的字母按照字母顺序进行排序。列表在内存的LIST到LIST +  $n - 1$ 中进行排序，每个字符用ASCII码表示，占一个字节。 $n$ 值是一个16位数，保存在地址N处。

用C语言重写该任务的程序如图3-34a所示，它在68000上的实现由图3-34b中给出。该程序与图2-34b中的程序非常相似，只有如下一些细微的不同。

68000中的MOVE指令允许原操作数和目标操作数都可以在内存中。因此，当两项进行交换时，LIST( $k$ )中的值直接复制到LIST( $j$ )中。这就省去了在图2-34中的临时寄存器R4的使用，使得程序指令有一些少量的调整。

```

for (j = n - 1; j > 0; j = j - 1)
{
    for (k = j - 1; k >= 0; k = k - 1)
    {
        if (LIST[k] > LIST[j])
        {
            TEMP = LIST[k];
            LIST[k] = LIST[j];
            LIST[j] = TEMP;
        }
    }
}

```

a) C语言的排序程序

	MOVEA.L	#LIST,A1	表的开始处的指针
	MOVE	N,D1	初始化外部循环变
	SUBQ	#1,D1	址j到D1中
OUTER	MOVE	D1,D2	初始化外部循环变
	SUBQ	#1,D2	址k到D2中
	MOVE.B	(A1,D1),D3	当前最大值在D3中
INNER	CMP.B	D3,(A1,D2)	如果LIST(k) < [D3],
	BLE	NEXT	不交换
	MOVE.B	(A1,D2),(A1,D1)	交换LIST(k)和LIST
	MOVE.B	D3,(A1,D2)	(j)并将新的最大值载
	MOVE.B	(A1,D1),D3	入D3中
NEXT	DBRA	D2,INNER	计数器k和j减1, 如
	SUBQ	#1,D1	果没结束, 转移返回
	BGT	OUTER	

b) 用68000实现的程序

153

图3-34 68000字节排序程序

另一个不同就是在变址k变化到0的时候使用DBRA指令终止内部循环。注意, 由于j的最终值是1而不是0, 所以在外部循环中不能使用DBRA指令。

### 3.15.3 链表的插入和删除子程序

图3-35给出了在链表中插入操作的68000子程序, 该程序与图2-37中的程序完全相同。注意比较指令CMPA是用来比较地址值的, CMP是用来比较数据值的。

从链表中删除一条记录的68000程序如图3-36所示。该程序直接与图2-38中的程序相对应。

与图2-37和图2-38中的普通子程序一样, 图3-35中的插入子程序假定新记录的ID值与链表中现有的记录不相同, 图3-36中的删除子程序假定在链表中存在一个ID值与RIDNUM相同的记录。习题中的3.49和3.50考虑的是当假设条件不成立时如何修改子程序产生报错信息。

154

## 部分III IA-32 Pentium 实例

Intel公司在它的处理器产品系列中使用了Intel体系结构(IA)这个通用的名称。我们将描述具有32位内存地址和32位数据操作数的IA处理器, 也就是指IA-32处理器, 最新的是Pentium系列。第一款IA-32处理器是1985年面世的80386。其后, 80486 (1989)、Pentium (1993)、Pentium Pro (1995)、Pentium II (1997)、Pentium III (1999) 和Pentium 4 (2000) 相继问世。这些处理器是通过改进许多体系结构和微电子技术, 逐渐提升它们的性能级别。IA系列的发展过程我们将在

第11章中介绍。该系列中的最新成员具有专门用来处理多媒体图像信息和向量数据的指令。这些指令集的特征在这里只做简单的描述，在第11章中还有这些方面的阐述。IA-32指令集是非常庞大的。关于IA-32指令集体系结构和汇编语言的详细信息可以在Intel网站<sup>[8]</sup>以及Brey<sup>[9]</sup>、Dandamudi<sup>[10]</sup>和Tabak<sup>[7]</sup>的书中找到。

155

子程序			
INSERTION	CMP A.L	#0,A0	A0是RHEAD
	BGT	HEAD	
	MOVEA.L	A1,A0	A1是RNEWREC
HEAD	RTS		
	CMP.L	(A0),(A1)	新记录的ID与表头比较
	BGT	SEARCH	
	MOVE.L	A0,4(A1)	新记录成为新表头
SEARCH LOOP	MOVEA.L	A1,A0	
	RTS		
	MOVEA.L	A0,A2	A2是RCURRENT
	MOVEA.L	4(A2),A3	A3是RNEXT
	CMP A.L	#0,A3	
	BEQ	TAIL	
	CMP.L	(A3),(A1)	
	BLT	INSERT	
	MOVEA.L	A3,A2	移向下一记录
INSERT TAIL	BRA	LOOP	
	MOVE.L	A2,4(A1)	
	MOVE.L	A1,4(A2)	
	RTS		

图3-35 在链表中插入一条记录的68000子程序

子程序			
DELETION	CMP.L	(A0),D1	D1是RIDNUM
	BGT	SEARCH	
	MOVEA.L	4(A0),A0	删除头记录
	RTS		
SEARCH LOOP	MOVEA.L	A0,A2	A2是RCURRENT
	MOVEA.L	4(A2),A3	A3是RNEXT
	CMP.L	(A3),D1	
	BEQ	DELETE	
	MOVEA.L	A3,A2	
DELETE	BRA	LOOP	
	MOVE.L	4(A3),D2	D2是RTEMP
	MOVE.L	D2,4(A2)	
	RTS		

图3-36 从链表中删除一条记录的68000子程序

### 3.16 寄存器和寻址方式

在IA-32体系结构中，内存采用字节可寻址的32位地址，指令的操作数是8位或32位的。这些指令在Intel术语中称为“字节”和“双字”。16位的操作数在早期的16位Intel处理器中称为“字”。这里使用的是在2.2.2节中描述的little-endian寻址方式。多字节数据的操作数可以在任意的字节地

址处开始，没有必要必须限制在内存的特定地址边界上对齐。

3.16.1 IA-32寄存器结构

图3-37中所示的是处理器寄存器。虽然有一些例外，8个32位标有R0到R7的寄存器是通用寄存器，可以用来保存数据操作数或寻址信息。有8个浮点寄存器用来存储双字或四字（64位）的浮点操作数。浮点寄存器中还有一个总长度是80位的扩展字段，在图3-37中没有给出。第6章中对浮点数的表示和操作进行了讨论。这个主题在第3章中不再讨论。

IA-32体系结构是以具有不同用途的被标为段的不同内存区域为基础的。代码段保存程序的指令。堆栈段存放处理器堆栈，四个数据段是用来保存数据操作数的。图3-37中的6个段寄存器保存着用于在内存空间中确定这些段位置的选择器值。这些寄存器的具体功能将在第11章的IA系列讨论中进行阐述。这里，我们不必知道细节。IA-32体系结构中的32位地址假定是可以在程序中访问存储器的任何一个单元、处理器堆栈或数据堆区。

图3-37底部给出的两个寄存器是指令指针，它们作为程序计数器保存下一条将要执行的指令

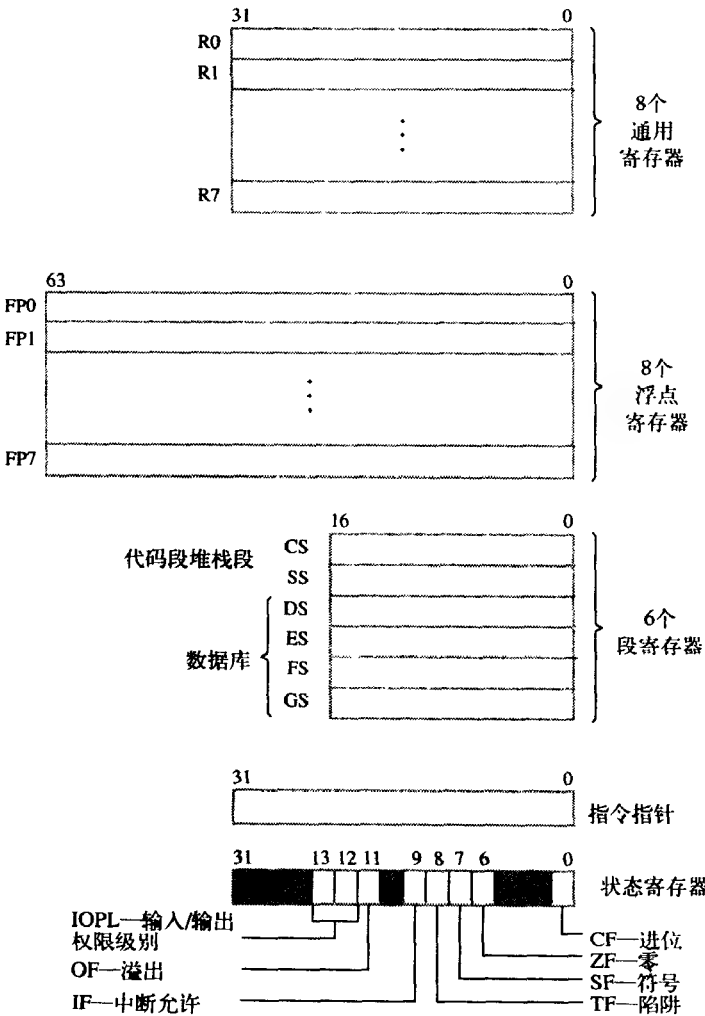


图3-37 IA-32寄存器结构



地址和保存条件码标志（CF、ZF、SF、OF）的状态寄存器。这些标志中包含有关算术操作的结果信息，就像在3.19节中讨论的一样。程序执行模式位（IOPL、IF、TF）与第4章讨论的输入/输出操作和中断的内容相关。

IA-32通用寄存器保持了与8位和16位处理器寄存器的兼容性。在那些处理器中，程序中对不同处理器的具体用法有一些应用上的限制。图3-38给出了IA-32寄存器与早期处理器寄存器之间的结合。8个通用寄存器分成三种不同的类型：数据寄存器用来保存操作数，指针和变址寄存器用来存储地址和用来确定内存操作数有效地址的变址地址。

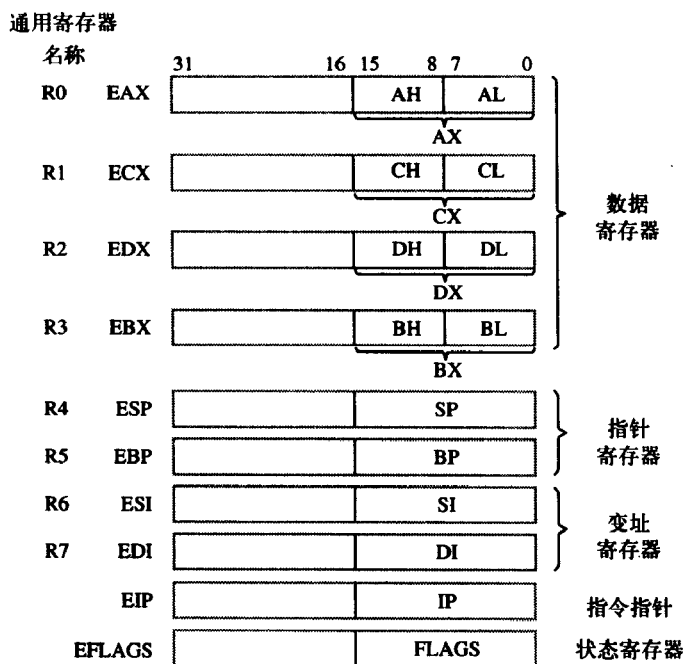


图3-38 IA-32与早期的Intel处理器寄存器结构的兼容性

在Intel最初的8位处理器中，数据寄存器称为A、B、C和D。在随后的16位处理器中它们被标志为AX、BX、CX和DX。每个寄存器中的高位字节和低位字节分别用H和L来标识。例如，在寄存器AX中的两个字节是AH和AL。IA-32处理器中，前缀E用来表示相应“扩展的”32位寄存器：EAX、EBX、ECX和EDX。E前缀标志还可以用于其他的32位寄存器，如图3-38中所示。它们是早期处理器的16位寄存器的相应版本。

这种寄存器标志在Intel技术文档和其他Intel处理器描述中使用。由于保持历史标志的原因，Intel在它的处理器系列中必须保持着向上兼容的特点。也就是说为早期的16位处理器编写的机器语言程序，只要是IA-32处理器的状态设置正确，就可以无需修改在IA-32处理器上正确运行。我们将在给定的汇编语言程序例子中，使用E前缀寄存器标志，因为IA-32处理器的汇编语言的当前版本中使用的就是这些助记符。当字节操作数在相应的32位寄存器中的低八位时，将使用AL、BL等标志。

在程序执行由使用指令前缀字节指定的指令期间，IA-32处理器状态可以在32位和16位操作之间进行动态切换。这一特点我们将在第11章中讨论。

### 3.16.2 IA-32 寻址方式

IA-32体系结构有一套大而灵活的寻址方式。它们既可以访问个别数据项，也可以访问从指定内存地址开始的有序表数据项。我们给出这些方式的完整定义以及在汇编语言中的表达方式。

在2.5节中已描述过，大多数处理器中都有基址方式。它们是：立即方式、绝对方式、寄存器方式和寄存器间接方式。Intel对绝对方式使用了直接来描述，所以在这里我们还采用同样的方式。还有几种用来访问内存数据操作数的更加灵活的寻址方式。在2.5节中描述的最灵活的方式就是具有通用符号 $X(R_i, R_j)$ 的变址方式。操作数的有效地址EA计算如下

$$EA = [R_i] + [R_j] + X$$

其中 $R_i$ 和 $R_j$ 是通用寄存器， $X$ 是一个常量。寄存器 $R_i$ 和 $R_j$ 分别称为基址和变址寄存器，常量 $X$ 叫做位移量。IA-32寻址方式中包括这种方式和该类型的更简单变化形式。

整个IA-32寻址方式定义如下：

**立即方式**——操作数包含在指令中。它是一个8位或32位的数，其长度用指令的操作码中的一位来指定。该位是0表示短版本，1表示长版本。

**直接方式**——操作数的内存地址是通过指令中的一个32位值给出。

**寄存器方式**——操作数包含在指令中的一个通用寄存器中。

**寄存器间接方式**——操作数的内存地址包含在指令中的一个通用寄存器中。

**带有位移量的基址方式**——在指令中给出了一个8位或32位的带符号的位移量以及一个用作基址寄存器的通用寄存器。操作数的有效地址是基址寄存器和位移量的和。

**带有位移量的变址方式**——在指令中给出一个32位的带符号的位移量，一个用作变址寄存器的通用寄存器和一个1、2、4或8的比例因子。操作数的有效地址是变址寄存器的内容乘以比例因子之后再与位移量相加的和。

**带有变址的基址方式**——在指令中给出两个通用寄存器和一个1、2、4或8的比例因子。寄存器用作基址和变址寄存器，操作数的有效地址按如下方式计算：变址寄存器的内容与比例因子相乘，之后再与基址寄存器中的内容相加。

**带有变址和位移量的基址方式**——在指令中给出一个8位或32位的带符号的位移量，两个通用寄存器和一个1、2、4或8的比例因子。寄存器用作基址和变址寄存器，操作数的有效地址按照如下方式计算：变址寄存器中的内容乘以比例因子，结果再与基址寄存器的内容及位移量相加。

159

IA-32寻址方式与汇编语言表达方式在表3-3中给出。在表中还给出了操作数有效地址的计算。正如中表3-3的脚注中说明的那样，寄存器ESP不能用作变址寄存器，它是作为处理器堆栈指针的。现在，我们给出几个例子来说明如何使用寻址方式来访问操作数。

在双操作数的指令中，源操作数（src）和目标操作数（dst）使用下面的汇编语言来指定：

OPcode dst, src

这个顺序与本章部分I的ARM体系结构是相同的，但却和第2章及本章部分II Motorola 68000处理器中的用法相反。例如，指令

MOV dst, src

执行的操作

$dst \leftarrow [src]$

表3-3 IA-32寻址方式

名 称	汇 编 语 法	寻 址 功 能
立即方式	Value	Operand = Value
直接方式	Location	EA = Location
寄存器方式	Reg	EA = Reg 即Operand = [Reg]
寄存器间接方式	[Reg]	EA = [Reg]
带有位移量的基址方式	[Reg + Disp]	EA = [Reg] + Disp
带有位移量的变址方式	[Reg*s + Disp]	EA = [Reg] × S + Disp
带有变址的基址方式	[Reg1 + Reg2*S]	EA = [Reg1] + [Reg2] × S
带有变址和位移量的基址方式	[Reg1 + Reg2*S + Disp]	EA = [Reg1] + [Reg2] × S + Disp

Value = 8位或32位带符号数

Location = 32位地址

Reg, Reg1, Reg2 = 通用寄存器 EAX、EBX、ECX、EDX、ESP、EBP、ESI、EDI中的一个，其中的ESP不能用作变址寄存器

Disp = 8位或32位带符号数，除了在带有位移量的变址方式中只能使用32位

S = 比例因子1、2、4或8

160

使用Move指令来说明IA-32寻址方式是非常方便的。指令

MOV EAX, 25

使用立即寻址方式将十进制数值25传送到EAX寄存器中。用0到9这种数字形式指明数值是十进制计数方式。后缀B和H分别用来表示二进制和十六进制数。例如，指令

MOV EAX, 3FA00H

将十六进制数3FA00传送到EAX中。

指令

MOV EAX, LOCATION

使用直接寻址方式将内存中地址标志LOCATION处的双字传送到寄存器EAX中。这里是假定在汇编语言程序的数据声明部分中已经定义了这个内存地址标志。我们将在3.18节中了解这是如何实现的。如果LOCATION代表的地址是1000，那么这条指令将1000处的双字传送到EAX中。

在IA-32汇编语言程序中对立即寻址方式和直接寻址方式之间的区别作了一些讨论，因为有时会产生一些混淆。考虑下面的情况，有时对用作立即操作数的数字常量给出符号名是非常方便的。汇编指示

NUMBER EQU 25

将十进制数25与一个符号名NUMBER关联起来，正如在2.6.1节中讨论的一样。如果这样做，那么指令

MOV EAX, NUMBER

将被汇编程序解释成NUMBER是一个立即操作数，要被传送到寄存器EAX中。相反，如果NUMBER定义成一个地址标志，这个指令将按照直接寻址方式来解释。

在很多汇编语言中，这种潜在的混淆状态用一个特殊的符号来避免，例如，#作为前缀来说明是立即寻址方式。在IA-32汇编语言中，使用方括号明确地指明直接寻址方式，如指令

MOV EAX, [LOCATION]

然而, 如果LOCATION已经定义成一个地址标志时, 就不必使用方括号了。

当有必要将一个地址标志处理成一个立即操作数时, 使用汇编指示OFFSET。例如, 指令

MOV EBX, OFFSET LOCATION

161

采用立即寻址方式, 将地址标志LOCATION (例如是1000) 处的值传送到EBX寄存器中。然后, EBX寄存器可以当作寄存器间接方式在指令

MOV EAX, [EBX]

中将内存地址LOCATION处的内容 (包含在寄存器EBX中) 传送到寄存器EAX中。OFFSET是一个汇编指示, 它用来表示一个地址, 该地址总是保存着从内存段的起始位置到包含该地址之间的一个相对距离。在所有这些例子中, 寄存器方式常常用来说明目标。

这些例子已经说明了基本的寻址方式: 立即、直接、寄存器和寄存器间接方式。其余的四种寻址方式在访问内存操作数时提供了更加灵活的方式。

带有位移量的基址方式在图3-39a中作了说明。寄存器EBP用作基址寄存器。双字操作数与基址寄存器1000相距60个字节, 也就是在地址1060处可以使用指令

MOV EAX, [EBP + 60]

传送到寄存器EAX中。

IA-32指令和寻址方式可以对字节操作数和双字操作数进行处理。例如, 还是假定基址寄存器EBP中的地址是1000, 在地址1010处的字节操作数可以使用下面的指令

MOV AL, [EBP + 10]

装入到EAX寄存器中的低字节部分中。由于目标操作数AL是EAX寄存器的低字节部分, 所以汇编程序选择用作字节数据的Move操作码版本。

最灵活的寻址方式就是带有变址和位移量的基址方式。图3-39b中给出了一个例子, 其中用EBP和ESI作为基址和变址寄存器。这个例子给出的是如何使用这种方式在一个双倍字长的操作数表中访问一个双倍字长的操作数。该表是在与基地址1000距离200处开始。对变址寄存器中的内容使用比例因子4, 可以使用变址寄存器中连续的变址0, 1, 2, ...来访问在地址1200, 1204, 1208...处的连续双倍字操作数。图中的例子表示, 当变址寄存器中的值是40时, 就是要访问地址1360处的双倍字 (也就是 $1000 + 200 + 4 \times 40$ )。这个操作数通过指令

MOV EAX, [EBP + ESI \* 4 + 200]

装入到寄存器EAX中。在这个寻址方式中使用比例因子4, 可以使在一个循环程序中访问表中连续的双倍字操作数更加容易, 只需在每遍循环中对变址寄存器加1。在对这两种方式作了比较详细的讨论之后, 与其紧密相关的带位移量的变址方式和带变址的基址方式将会很容易理解。

162

在结束寻址方式的讨论之前, 表3-3中的各项注释是很有用的。带有位移量的基址方式看起来是多余的, 由于通过带有位移量的变址方式 (比例因子是1), 就可以实现同样的效果。但是由于前一种方式可以编码在一个字节之中, 其还是有用的。此外, 带有位移量的变址方式的位移量只能是32位的。

关于寻址方式如何编码成机器指令的讨论包含在下一节中。更多细节参见附录D。

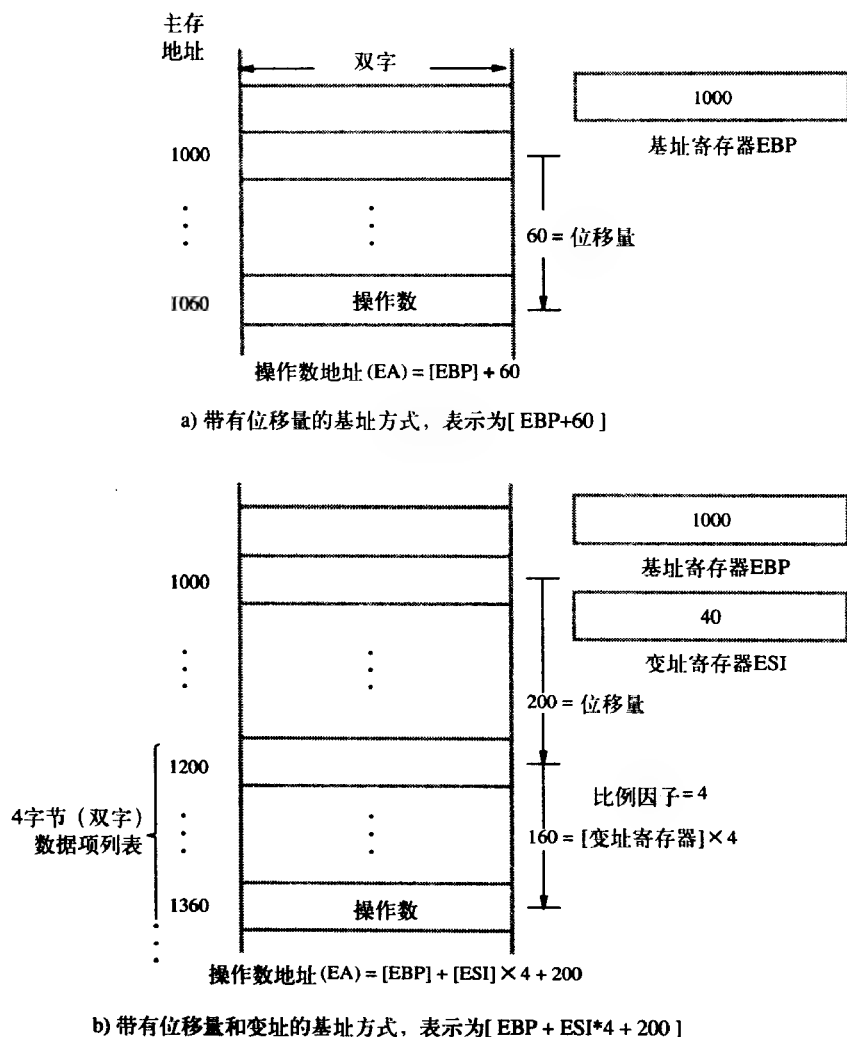


图3-39 IA-32结构寻址方式举例

### 3.17 IA-32 指令

IA-32指令系统是很庞大的。它可以按照动态长度指令格式进行编码, 而无需使用完全规则的编码方案。我们将在3.17节中来了解它的指令格式。大多数的IA-32指令都有一个到两个操作数。在双操作数的情况下, 只能有一个操作数在内存中。另一个操作数必须是在处理器的寄存器中。除了通常的内存与处理器寄存器之间的数据传送指令以外, 还有执行算术操作的指令, 指令系统中包含很多不同的对数据的逻辑和移位/循环移位操作指令。字节串指令用来对非数值数据进行处理。在指令系统中还直接支持对处理器堆栈进行的压栈和弹出操作。

我们将对指令系统中的一小部分进行介绍, 以此说明它们在一个简单完整的程序中是如何使用的。指令

ADD dst, src

执行的操作是

$$\text{dst} \leftarrow [\text{dst}] + [\text{src}]$$

正如我们在前面看到的那样，

```
MOV dst, src
```

执行的操作

$$\text{dst} \leftarrow [\text{src}]$$

假定两个操作数在寄存器EAX和EBX中，它们的和可以用下面的两条指令序列在EAX中被计算出来并存储在内存单元SUM中：

```
ADD EAX, EBX
```

```
MOV SUM, EAX
```

由于在双操作数指令中只能有一个操作数放在内存中，操作

$$C \leftarrow [A] + [B]$$

164

中包括了三个存储单元A、B和C，所以可以用指令序列：

```
MOV EAX, A
```

```
ADD EAX, B
```

```
MOV C, EAX
```

来执行。减法指令

```
SUB dst, src
```

执行操作

$$\text{dst} \leftarrow [\text{dst}] - [\text{src}]$$

非常有用的单操作数指令INC和DEC可以对它们的操作数进行加1或减1。

在给出一个完整的数值相加循环程序之前，我们还需要一些指令。其中之一就是条件转移指令。转移指令

```
JG LOOPSTART
```

如果最近的算术操作结果大于0，转移指令就执行在内存LOOPSTART处的指令。所有的条件转移指令都是以字母J（表示跳转）开头，后面跟着的字符指定条件。在本例中，G代表大于0。其他的条件转移将在后面讨论。

为了在寄存器间接寻址中使用通用寄存器，首先必须将内存地址装入到要访问的寄存器中。IA-32指令系统中提供了两种实现方式。如果地址是作为地址标志明确知道的，比如说LOCATION，它可以在一个传送指令中用立即寻址方式将其装到寄存器中，比如

```
MOV EBX, OFFSET LOCATION
```

该指令将标志LOCATION所代表的地址装入到寄存器EBX中。另一种方式是使用一条称为“装入有效地址”（助记符是LEA）的指令。指令

```
LEA EBX, [EBP + 12]
```

将[EBP+12]处的操作数地址装到寄存器EBX中。当指令执行时，这个地址要依赖于寄存器EBP中的内容。

165

### 用于数值相加的循环程序

采用刚刚介绍过的指令，现在我们可以利用一个循环给出数值相加的程序。假定在内存以

NUM1开始处有一个列表，表中的一个32位整数保存内存位置N中。汇编语言程序如图3-40a所示可以将数值相加并将它们相加的和保存到内存的SUM中。

寄存器EBX中保存着地址值NUM1。在STARTADD（循环指令中的第一条指令）单元指令中的带变址的基址寻址方式中，它当作基址寄存器，寄存器EDI当作变址寄存器。在进入循环之前，它被清成0。在第一次循环中，[EBX]=NUM1处的第一个数被加到初始化成0的EAX寄存器中。之后，变址寄存器增加1。这样，在循环的第二遍中，加法指令中的比例因子4将使得在地址NUM1+4字节处的第二个32位数加到EAX中。在后续的循环中，分别将NUM1+8, NUM1+12, ...处的数相加。寄存器ECX用作计数器寄存器。该寄存器最初是由程序中的第二条指令将内存中N单元中的内容装载到其中，在程序中的每遍循环中都减1。

	LEA	EBX,NUM1	初始化基址寄存器 (EBX) 和计
	MOV	ECX,N	数寄存器 (ECX)
	MOV	EAX,0	清除累加器 (EAX) 和变址寄存
	MOV	EDI,0	器 (EDI)
STARTADD:	ADD	EAX,[EBX + EDI * 4]	将下一个数加到EAX
	INC	EDI	变址寄存器增1
	DEC	ECX	计数寄存器减1
	JG	STARTADD	如果[ECX]>0, 转移返回
	MOV	SUM,EAX	将和保存到内存中

a) 直接方法

	LEA	EBX,NUM1	载入基址寄存器EBX并调整保存
	SUB	EBX,4	NUM1-4
	MOV	ECX,N	初始化计数/变址寄存器 (ECX)
	MOV	EAX,0	清除累加器 (EAX)
STARTADD:	ADD	EAX,[EBX + ECX * 4]	将下一个数加到EAX
	LOOP	STARTADD	ECX减1, 如果[ECX]>0转移返回
	MOV	SUM,EAX	将和保存到内存中

b) 更紧凑的程序

166

图3-40 数值相加的IA-32程序

条件转移指令JG在[ ECX ] > 0时引起转移返回。当ECX的内容达到0时，全部的数已经相加完毕。没有执行转移，而是用传送指令将寄存器EAX中相加的和保存到内存中的SUM单元中。

通过对图3-40a程序中的下面两条指令的观察，这个任务可以采用一个更简捷的方式编写。两条指令序列

```
DEC ECX
JG STARTADD
```

通常是出现在程序循环的最后部分。因此，IA-32指令系统中包括一个将这两条指令的操作合并成一条指令的功能。指令

```
LOOP STARTADD
```

首先将ECX寄存器减1，之后如果ECX的内容还没有达到0，就转移到目标地址STARTADD处。

观察的第二点就是使用了两个寄存器EDI和ECX作为计数器。如果反向扫描所要相加的数字列表，从列表的最后一个数开始，则只需使用一个计数寄存器。由于寄存器ECX是LOOP指令隐含引用的寄存器，所以我们使用该寄存器。假定[ N ] = n，当EDI中包含的序列值是0, 1, 2, ..., (n - 1)，

第一个程序用地址序列NUM1, NUM1 + 4, NUM1 + 8, ..., NUM1 + 4(n - 1) 来访问数值。当ECX中的值是 $n, n - 1, \dots, 1$ 时, 新程序如图3-40b所示, 使用的地址序列是 (NUM1 - 4) + 4 $n$ , (NUM1 - 4) + 4(n - 1), ..., (NUM1 - 4) + 4(1)。因此, 为了解决EDI与ECX中的顺序不同, 要将新程序的基址寄存器中的值从NUM1改写成NUM1 - 4。在新程序中的最后一遍循环中执行循环指令之前, [ECX] = 1, 最后要相加的数在内存中的NUM1处。

读者应该注意到在处理列表和数组中, 在以0或1开始的进行适当变址计算中, 以及在正确选择转移条件中, 会经常出现这种详细的推理类型。它也可能是一些微妙错误的根源所在。在高级语言中, 列表变量明确引用LIST (0), LIST (1), ..., LIST (n - 1), 以及循环范围与下面使用的表达式变址值相关:

FOR  $i$  FROM 0 UPTO (n - 1)

或

FOR  $i$  FROM (n - 1) DOWNT0 0

这种情况下困难相对来说要小一些。

167

这里使用加法循环程序的例子简要讨论了一些常用的IA-32指令, 并对使用的指令系统的基本特征和汇编语言作了简要介绍。现在, 我们来描述指令的机器表示格式。

### 机器指令格式

机器指令的一般格式如图3-41所示。指令是可变长的, 变化的范围从一个字节 (只有一个操作码, 通常情况下都需要) 到12个字节, 由四个字段构成。操作码字段是由一个或两个字节构成的, 大多数指令只需要一个字节。紧跟在操作码字段之后的寻址方式信息保存在一到两个字节中。对于只使用一个寄存器来产生操作数有效地址的指令, 寻址方式字段中只需要一个字节。对于表3-3中的最后两个寻址方式的编码是需要两个字节的。这些方式使用两个寄存器来产生内存操作数的有效地址。

在计算内存操作数的有效地址中要使用一个位移量, 那么这个位移量就用一个或四个字节进行编码, 并放在紧跟在寻址方式字段之后的字段中。如果操作数之一是立即数, 那么它被放在指令的最后一个字段中, 占用一个或四个字节。

对于一些简单的例如下面首先要讨论的这些指令, 指令中包括的一个寄存器的代码, 在操作码字节中给出。但是, 对于大多数指令和寻址方式而言, 使用的寄存器是在寻址方式字段中指定的。

在任何指令系统中, 指令用可变长格式进行编码, 那么指令的位模式在从左向右读取时, 必须确定指令的总长度。这是因为程序中的连续指令在内存中是一个接着一个放置的, 没有其他的可用信息来指示指令间的边界, 所以总长度是很重要的。

#### 单字节指令

寄存器可以用只占一个字节的指令进行增加或减小。例如

INC EDI

和

DEC ECX

其中通用寄存器EDI和ECX是在单个操作码字节中用3位码指定的。



操作码	寻址方式	位移量	立即数
1或2 字节	1或2 字节	1或4 字节	1或4 字节

图3-41 IA-32指令格式

168

**立即方式编码**

操作码说明何时使用立即寻址方式。例如，指令

```
MOV EAX, 820
```

被编码成5个字节。单字节操作码说明是传送操作，使用的是32位的立即操作数以及目标寄存器的名称。操作码字节后面紧跟着4字节的立即数的值820。当使用8位的立即操作数时，指令如下：

```
MOV DL, 5
```

对该指令编码仅需要两个字节。

**寻址方式和位移量字段**

作为一般的规则，双操作数指令中的一个操作数必须是放在寄存器中的。另一个操作可以在寄存器中也可以在内存中。有两个例外情况，两个操作数都可以在内存中。第一种情况是源操作数是立即操作数，目标操作数在内存中。第二种情况是在处理器堆栈上的压入和弹出操作指令。堆栈位于内存的堆栈段中，它可以将一个内存操作数压入堆栈或从堆栈中将一个操作数弹出到内存中。我们将在后面的3.22节中讨论这个内容。

当两个操作数都在寄存器中时，只需要一个寻址方式字节。例如，指令

```
ADD EAX, EDX
```

被编码成两个字节。第一个字节包含操作码，另一个是寻址方式字节，用来指定两个寄存器。

现在，让我们考虑几个指令编码，其中一个操作数在寄存器中，另一个操作数在内存中。图3-40程序中的指令

```
MOV ECX, N
```

被编码成六个字节：一个字节是操作码；一个字节是寻址方式字节，它指定直接方式以及目标寄存器ECX；四个字节用于内存单元N的地址。

该程序中的指令

```
ADD EAX, [EBX + EDI * 4]
```

由于使用了两个寄存器来产生源操作数的有效地址，所以寻址方式字段的字节需要两个字节。比例因子4包含在这两个字节中的第二个字节中。这样，指令全长需要三个字节，包括操作码字节。

第三个例子来考虑指令

169

```
MOV DWORD PTR [EBP + ESI * 4 + DISP], 10
```

汇编指示DWORD PTR用来说明立即操作数10的长度是32位的。在其他的汇编程序中，操作数长度的说明通常是包含在操作码助记符中的。例如，在本章部分II中讨论的Motorola 68000，MOVE.B指定的是单字节的操作数，MOVE.L指定的是4字节的长字操作数。如果使用的是32位的位移量DISP，那么这条指令的编码需要11个字节：一个字节用于操作码；两个字节用于寻址方式字段；每个位移量和立即数字段都需要四个字节。在表3-3中说明了位移量的长度可以是8位

或32位的。在寻址方式的两个字节中的第一个字节里指定了长度。

在双操作数指令的编码中，寄存器操作数和内存操作数是按照固定的顺序说明的。首先说明的总是寄存器操作数。为了将指令

```
MOV EAX, LOCATION
```

(将内存中LOCATION处的内容装入到寄存器EAX中)和指令

```
MOV LOCATION, EAX
```

(将寄存器EAX的内容存储到LOCATION中)进行区别，在操作码字节中包含一位称为方向位的特殊位，这个位用来指明哪个操作数是源操作数。

在IA-32体系结构中，操作码和寻址方式的编码稍微有些复杂，并且还有很多的不一致和例外的情况。虽然这使得编译器充分利用指令系统和寻址方式的全部特征有些困难，但是这也无疑是IA-32体系结构非常强大和灵活的特征。

附录D中给出了IA-32指令的总结，以及在个人计算机上输入和运行汇编语言程序的指南。

### 3.18 IA-32汇编语言

通过图3-40中的程序，说明了IA-32汇编语言对于操作码、寻址方式和指令地址标志具体指明的基本情况。正如2.6.1节中描述的那样，汇编指示对定义一个程序的数据区域和数据单元符号名称及实际的物理地址值之间的通信是必要的。

图3-40b中程序的完整汇编语言程序如图3-42所示。.data和.code汇编指示定义了程序的数据和代码（指令）部分的起始位置。在数据区中，DD指示符分配了4字节的双倍字数据单元。NUM1是五个双倍字中的第一个字的地址标号，这五个双倍字初始化成十进制数17、3、-51、242和-113。下两个双倍字单元（初值为5和0）给出了地址标号N和SUM。

170

在代码区指令的寻址方式中，使用了三个在数据区中声明的符号名。MAIN标号用来说明指令执行的开始地址，该标号用在结束程序文本文件的END指示符所在的模块。其他的汇编指示，如在2.6.1节中讨论的EQU，也是可以使用的。

汇编指示	{	.data		
		NUM1	DD	17,3,-51,242,-113
		N	DD	5
		SUM	DD	0
		.code		
产生机器指令的语句	{	MAIN :	LEA	EBX, NUM1
			SUB	EBX, 4
			MOV	ECX, N
			MOV	EAX, 0
		STARTADD :	ADD	EAX, [EBX+ECX * 4]
			LOOP	STARTADD
			MOV	SUM, EAX
汇编指示			END	MAIN

图3-42 完全采用IA-32汇编语言编写的图3-40b中的程序

### 3.19 程序流控制

有两种主要方法可以改变指令的直线执行顺序流程。调用子程序并从子程序返回可以打破这种直线执行，这些要在3.22节中介绍。还有转移指令，无论是条件转移还是无条件转移，也可以改变这种流程。我们现在讨论转移指令。在IA-32的术语中，转移指令称为跳转（Jump）。

#### 3.19.1 条件跳转及条件码标志

我们将图3-40a中的指令

JG STARTADD

**[171]** 作为条件跳转指令的例子。条件是“大于0”，在条件码中用G后缀来表示。这个条件与刚刚执行完的数据操作指令的结果有关，在本例中该指令是

DEC ECX

指令（如递减、加法或其他算术运算和比较操作指令）生成的结果属性，记录在处理器状态寄存器中的四个条件码标志位中，如图3-37所示。这些属性称为SF（符号）、ZF（零）、OF（溢出）和CF（进位）标志，与在2.4.6节中描述的方式一样进行置1或清0。它们分别称为N、Z、V和C，但有一个例外。对于减法操作，CF在没有进位时候被设置成1，表示借位信号。后面的条件跳转指令可以对标志位进行检测，以此来确定是否执行跳转操作。在我们的例子中，如果条件[ECX]>0，执行控制将切换到跳转目标地址STARTADD处的指令执行。

条件跳转指令中并不包含跳转目标地址是绝对地址的情况，而是包含一个带符号数，将该数与指令指针寄存器中的内容相加来确定目标地址。这样，目标地址与指令指针中的地址相关。在一条指令取出后执行的第一步操作是将指令指针提前指向下一条指令。因此，在跳转目标地址与相对偏移量相加时，指令指针中包含的是紧跟在跳转指令之后的指令地址。在我们的例子中，假定地址STARTADD是1000。图3-40a中需要编码的四条指令ADD、INC、DEC和JG的总字节数是7个字节。指令指针寄存器EIP的更新内容是1007，也就是程序中的最后MOV指令的地址。因此，到跳转目标地址的相对地址是-7，也就是保存在条件跳转指令中的值。这个小负数可以用一个字节表示。所以，当相对目标地址是在-128到+127之间时，包括操作码字节需要编码的条件跳转指令只需要两个字节。当使用4字节的偏移量时，跳转的目标地址范围会更大。

在本例中，要检测ECX寄存器减小后的结果，看它是否是大于0。结果的其他算术属性可以用不同的条件跳转指令来检测。例如，对于带有操作码JZ（或JE）和JS的指令，分别在等于0和符号位是1（负数）的时候进行跳转。

#### 比较指令

程序中要经常根据比较两个数值的结果进行条件跳转。比较指令

CMP dst, src

执行的操作是

[dst] - [src]

根据获得的结果来设置条件码标志。但这时任何一个操作数都不会改变，总是第一个操作数与第二个操作数进行比较。例如，如果通过一个“大于”的条件指令来跟踪比较指令，那么希望如果目标操作数大于源操作数时就跳转到目标地址处。

**[172]**

### 3.19.2 无条件跳转

一个无条件跳转指令JMP是一定要产生转移到目标地址处的指令。除了使用短（一个字节）或长（四个字节）的相对符号偏移量来确定目标地址以外，像在条件跳转指令中一样，JMP指令也可以使用其他的寻址方式，这种灵活性在产生目标地址中是非常有用的。考虑在很多高级语言中的Case语句。在程序中的某处，需要在很多的选择计算中执行一个正确的计算，其中每一个计算涉及一种情况。假定这些情况中每一个程序的第一条指令的4字节地址保存在内存中的一张表中，表的起始位置是JUMPTABLE。如果这些情况是采用0, 1, 2, …进行编号的，要执行情况（Case）的变址保存在寄存器ESI中，那么通过执行指令

JMP [JUMPTABLE + ESI \* 4]

就完成到选择情况的跳转操作，其中使用的是带位移量的变址寻址方式。

## 3.20 逻辑和移位/循环移位指令

### 3.20.1 逻辑操作

IA-32体系结构中具有执行逻辑与、或和异或操作的指令。这种指令对两个操作数进行位操作，将其结果保存在目标地址中。例如，假设在寄存器EAX中保存着十六进制数0000FFFF，寄存器EBX中的值是02FA62CA。那么指令

AND EBX, EAX

将EBX中的左半部分全部清0，后半部分保持不变。结果放在EBX中为000062CA。

指令系统中还有一条NOT指令，对操作数的所有位进行逻辑求反，也就是将全部的1变成0，全部的0变成1。

### 3.20.2 移位与循环移位操作

利用逻辑移位或算术移位可以将一个操作数左移或右移，给出的数值是决定要移动的位数。移位指令的格式是

Opcode dst, count

其中要移位的目标操作数通过一般的寻址方式指定，count可以用一个8位的立即数给出，也可以放在一个8位寄存器CL中。共有四条移位指令：

SHL 逻辑左移

SHR 逻辑右移

SAL 算术左移（与SHL相同）

SAR 算术右移

移位操作已在2.10中进行了讨论并用图2-30举例做了说明。

还有四条循环移位指令。它们是针对不带进位标志CF的向左循环和向右循环移位指令ROL和ROR，带有CF的RCL和RCR循环移位指令。所有这四种操作已在图2-32做了说明。

#### 数字打包程序

作为这些指令的一个简单应用，考虑图2-31中的BCD数字打包程序，该程序的IA-32代码如图3-43所示。寄存器AL和BL中装有两个ASCII码字节。SHL指令将AL中的字节左移四位，低位

部分用0填充。在该指令中操作数字段的第二个数是表示要传送的位数。AND指令将第二个字节的高四位清0。最后，使用OR指令将所期望的4位形式的BCD码合并到AL中，之后保存到内存的字节单元PACKED中。

### 3.21 I/O操作

#### 3.21.1 存储器映射I/O

在现代的计算机系统中，对输入/输出设备缓冲寄存器访问的最常用方法是在2.7节中描述的存储器映射I/O方法。IA-32的传送指令可以将指示传送给I/O设备，并可以将数据和状态信息及设备之间进行来回传送。例如，假设键盘和显示设备具有同步标志SIN和SOUT（参见图2-19），它们分别保存在设备状态寄存器INSTATUS和OUTSTATUS中的第3位中。采用程序控制I/O，使用下面的等待循环，可以完成从键盘缓冲寄存器DATAIN中读入一个字节放到寄存器AL中：

```
READWAIT: BT    INSTATUS, 3
           JNC   READWAIT
           MOV   AL, DATAIN
```

指令BT是一个位检测指令。源操作数指定的目标位置值（在本例中就是立即数3）装入到进位标志CF中。条件跳转JNC（如果无进位就跳转）在CF=0时引起跳转到READWAIT处。

与此类似，输出操作将寄存器AL中的一个字节发送到显示缓冲寄存器DATAOUT中，实现如下：

```
WRITEWAIT: BT    OUTSTATUS, 3
           JNC   WRITEWAIT
           MOV   DATAOUT, AL
```

图3-44中给出的是一个IA-32程序，它从键盘上读取一行字符将其保存在以LOC为开始内存地址中，并将这些字符显示出来。该程序是模仿图2-20中的一般程序。寄存器EBP指向该行字符保存的内存区。

READ:	LEA	EBP,LOC	EBP指向内存区
	BT	INSTATUS,3	等待有字符输入到DATAIN中
	JNC	READ	
	MOV	AL,DATAIN	将字符传送到AL中
ECHO:	MOV	[EBP],AL	把字符保存到内存，并增加指针
	INC	EBP	
	BT	OUTSTATUS,3	等待显示就绪
	JNC	ECHO	
	MOV	DATAOUT,AL	向显示器发送字符
	CMP	AL,CR	如果不是回车，继续读取字符
	JNE	READ	

图3-44 读取一行字符并显示出来的IA-32程序

#### 3.21.2 独立I/O

175 IA-32指令系统中还有两条操作码是IN和OUT的指令，它们只能用于I/O。这些指令所产生的

地址在另一个地址空间中，它与其他指令使用的地址空间相互独立。这种安排称为独立I/O以区别于存储器映射I/O，而后一种的I/O设备可寻址单元与内存单元所用的地址空间是相同的。这两种地址空间都可以用在Intel处理器芯片同样的地址和数据线上，并使用一条输出控制线来说明当前指令正在引用的是哪个地址空间。

在字节可寻址的I/O地址空间中使用的是16位的地址。8位和32位的I/O设备操作数单元用来保存数据、状态和命令信息。输入指令使用的格式是

IN REG, DEVADDR

这里目标REG必须是寄存器AL或EAX，分别表示一个8位或32位的操作数的传递。指令中的最后字段包含8位的设备地址DEVADDR。相应的输出指令是

OUT DEVADDR, REG

由于地址空间是字节可寻址的，向处理器发送一个8位ASCII字符的键盘设备有字节地址为DEVADDR的数据缓冲寄存器，而它的8位状态寄存器在地址DEVADDR+1处。

整个的16位I/O地址空间范围是64K；它可以通过输入指令

IN REG, DX

中的DX寄存器来引用。像前面一样，REG必须是AL或EAX。16位的设备地址包含在DX寄存器中，它是EDX寄存器的低16位，数据传送的宽度是由REG操作数的大小来决定的。相应的输出指令是

OUT DX, REG

### 3.21.3 块传送

在处理器和I/O设备之间除了传送单个信息项的指令IN和OUT外，IA-32体系结构还有两个块传送I/O指令：REPINS和REPOUTS。它们在内存和I/O设备之间串行地传送一批数据项，每次传送一项。操作码中的S后缀表示字符串，REP前缀表示“逐项地反复传送，直到整块传送完毕”。指令本身并没有指定用来描述传送的参数，这些指定参数是隐含在处理器寄存器DX、EDI和ECX中的，内容如下：

DX 包含一个16位I/O设备地址。

EDI 包含一个32位内存块的起始地址。

ECX 包含的是要传送的数据项的数目。

操作码助记符中的后缀B或D表示数据项长度是一个字节或一个双字长。这样，REPINSB就是字节块传送，REPINS D是双字块传送。块传送指令操作如下：在每个数据项传送完之后，变址寄存器EDI将根据数据项的长度来增加1或4，ECX寄存器减1。传送将一直重复直到计数器寄存器ECX的内容为0为止。这条单指令的作用等价于使用寄存器ECX作为循环计数器的循环程序。

看一个例子，假设从一个磁盘存储设备向内存中传送一个具有128个双字的块。假定磁盘设备的可寻址数据缓冲寄存器中包含一个双字数据项以及I/O地址DISKDATA。要写进内存的数据块的开始地址是MEMBLOCK，计数器寄存器ECX初始化成128，指令序列

LEA EDI, MEMBLOCK  
MOV DX, DISKDATA

```
MOV ECX, 128
```

```
REPINSDB
```

可以用来完成传送。程序中假定MEMBLOCK已经定义为一个地址标号，并已使用EQU汇编指示将DISKDATA定义成表示16位的设备数据缓冲寄存器的地址。

这里讨论的是独立I/O是如何组织的，同时还指出一个多数据项的块传送是如何通过使用地址计数器（EDI）和数据项计数器（ECX）的单指令来完成的。

### 3.22 子程序

像2.9节中阐明的那样，处理器堆栈数据结构是为了便于处理子程序的进入和返回的。在IA-32体系结构中，寄存器ESP用作堆栈指针，指向处理器堆栈当前栈顶元素（TOS）。堆栈的增长方向是内存地址的低地址方向。这种安排与2.8节中讨论的相同，在图2-21和图2-22中进行了说明。堆栈的宽度是32位，也就是全部的堆栈元素都是双字长的。

有四条压栈和弹出指令。指令

```
PUSH src
```

将ESP减4，然后将src处的双倍字保存到内存中ESP指向的位置处。指令

177

```
POP dst
```

是相反的过程，释放被ESP指向的TOS处的栈顶双倍字，将其保存在dst处，然后ESP加4，这样有效地将栈顶（TOS）元素从堆栈中移出。这些指令中暗含着将ESP作为堆栈指针。源操作数和目标操作数采用IA-32寻址方式来指定。两条指令可以压入或弹出多个寄存器中的内容。指令

```
PUSHAD
```

可以将全部的八个通用寄存器EAX到EDI中的内容压入堆栈，指令

```
POPAD
```

将它们按照相反的顺序弹出。当POPAD到达保存在ESP中原来的栈顶时，它将丢弃这四个字节并把它们放入到ESP中，并将剩余的值继续弹出分别放到各自的寄存器中。这两条指令作为实现子程序的一部分，用于有效地保存和恢复全部寄存器中的内容。

图3-40a中的列表加法程序可以写成一个如图3-45所示的子程序，参数通过寄存器进行传送。调用程序将相加数的第一个数的内存地址NUM1装入到寄存器EBX中，相加数的个数保存在内存中的N处，被装入到寄存器ECX中。调用程序希望得到的列表数据相加的和通过寄存器EAX返回。这样，寄存器EBX、ECX和EAX就用来传送参数。寄存器EDI在子程序在执行加法时作为变址寄存器，所以它的内容必须在子程序中利用PUSH和POP指令进行保存和恢复。

子程序被下面的指令调用：

```
CALL LISTADD
```

首先将返回地址压入堆栈，然后转移到LISTADD处，子程序保存（压入）EDI之后堆栈的内容如图3-45b所示。返回地址是调用程序中紧跟在调用指令之后的MOV指令的地址。指令RET通过将栈顶元素弹出到指令指针（程序计数器）EIP中来将执行控制返回给调用程序。

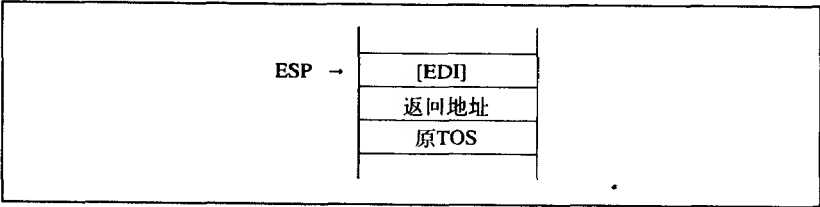
图3-46给出的是对图3-40a中的程序使用堆栈来传送参数而重新编写的一个子程序。参数

NUM1和n由调用程序中的PUSH指令压入堆栈，在执行完调用程序以后栈顶处于第二级。寄存器EDI、EAX、EBX和ECX在子程序中的用途与在图3-45中的相同。在子程序中的前八条指令是将它们的值进行保存并装入初始值和参数。这时，栈顶处于第三级。当数值由四个循环指令相加完以后，相加的和放到堆栈中，重写参数NUM1。在执行返回指令Return以后，调用程序中的ADD和POP指令将参数n从堆栈中移出并将相加的和弹出放到内存中的SUM处，恢复栈顶到第一级。

178

调用程序		
	⋮	
	LEA    EBX,NUM1	将参数载入EBX、ECX中
	MOV    ECX,N	
	CALL   LISTADD	转移到子程序
	MOV    SUM,EAX	将和保存到内存中
	⋮	
子程序		
LISTADD:	PUSH   EDI	保存EDI
	MOV    EDI,0	将EDI用作变址寄存器
	MOV    EAX,0	将EAX用作累加器
STARTADD:	ADD    EAX, [EBX + EDI * 4]	加下一个数
	INC    EDI	增加变址
	DEC    ECX	减小计数器
	JG      STARTADD	如果[ECX]>0转移返回
	POP    EDI	恢复EDI
	RET	转移返回调用程序

a) 调用程序和子程序



b) 子程序保存EDI之后的堆栈内容

图3-45 将图3-40a的程序重写为一个IA-32子程序；参数通过寄存器传送

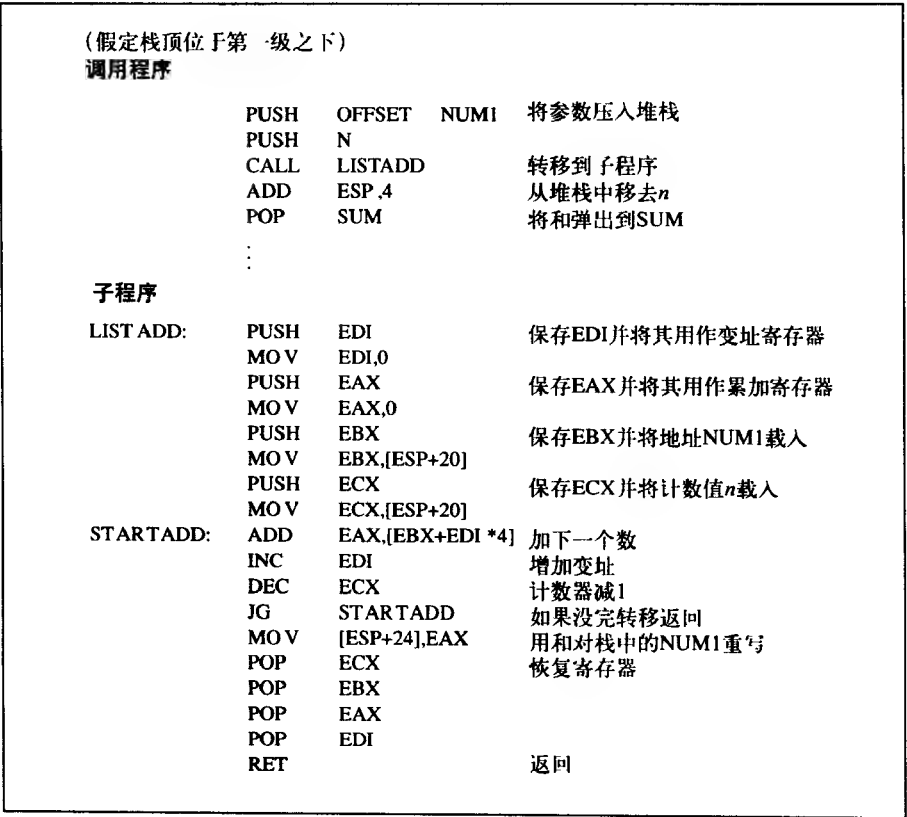
我们将要讨论的子程序的最后一个例子是处理嵌套调用的情况。图3-47给出了图2-28中程序的IA-32代码。在图3-48中给出了第一个和第二个子程序相应的堆栈结构。寄存器EBP用作结构指针，调用程序和子程序的结构与图2-28中的十分相近。图3-47中的不同是在保存和恢复寄存器时使用了多个PUSH和POP指令来代替图2-28中的多个传送指令。IA-32指令系统中有可以将全部的八个通用寄存器的内容压入栈或弹出栈的PUSHAD和POPAD指令，就像本节前面描述的那样，但是由于子程序只使用了寄存器集中的一半，所以我们在图3-47中选择使用单个PUSH和POP指令。

179

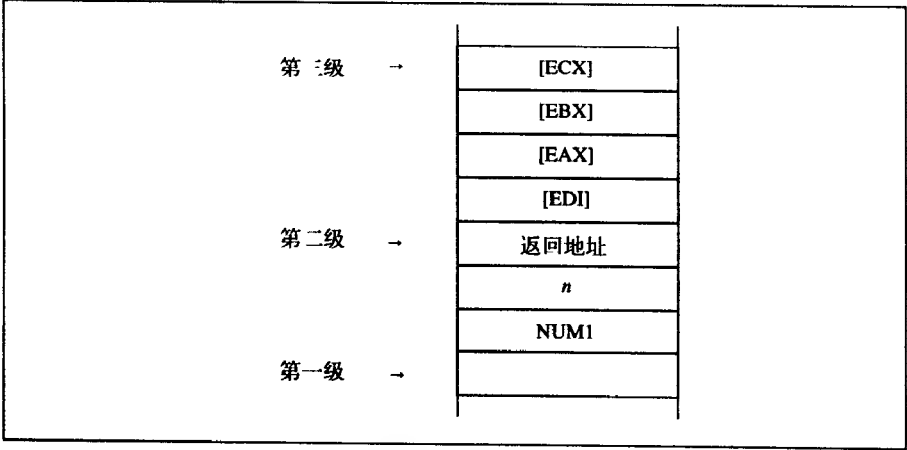
3.23 其他指令

前面我们描述的只是整个IA-32指令系统中的一小部分。这里将描述一些更重要的指令。





a) 调用程序和子程序



b) 不同时刻的堆栈结构

180

图3-46 将图3-40a中的程序重写为一个IA-32子程序；参数通过堆栈传送

3.23.1 乘法与除法指令

除了在3.17节中讨论的带符号整数的加法和减法指令以外，还有整数的乘法和除法指令，以及浮点数的乘除法指令。

地址	指令	注释
<b>调用程序</b>		
	⋮	
2000	PUSH PARAM2	将参数放入堆栈
2006	PUSH PARAM1	
2012	CALL SUB1	
2017	POP RESULT	保存结果
	ADD ESP,4	恢复堆栈级别
	⋮	
<b>第一个子程序</b>		
2100 SUB1:	PUSH EBP	保存结构指针寄存器
	MOV EBP,ESP	载入结构指针
	PUSH EAX	保存寄存器
	PUSH EBX	
	PUSH ECX	
	PUSH EDX	
	MOV EAX,[EBP+8]	取得第一个参数
	MOV EBX,[EBP+12]	取得第二个参数
	⋮	
	PUSH PARAM3	将参数放入堆栈中
2160	CALL SUB2	
2165	POP ECX	将SUB2结果弹出放入ECX
	⋮	
	MOV [EBP+8],EDX	将应答放入堆栈
	POP EDX	恢复寄存器
	POP ECX	
	POP EBX	
	POP EAX	
	POP EBP	恢复结构指针寄存器
	RET	返回到主程序
<b>第二个子程序</b>		
3000 SUB2:	PUSH EBP	保存结构指针寄存器
	MOV EBP,ESP	载入结构指针
	PUSH EAX	保存寄存器
	PUSH EBX	
	MOV EAX,[EBP+8]	取得参数
	⋮	
	MOV [EBP+8],EBX	将SUB2结果放入堆栈
	POP EBX	恢复寄存器
	POP EAX	
	POP EBP	恢复结构指针寄存器
	RET	返回到第一个子程序

图3-47 用IA-32汇编语言编写的嵌套子程序

181

通常，两个32位整数相乘会生成一个双倍长的64位乘积。然而，在很多应用中，乘积被认为是一个单倍长度的32位结果。对于这两种情况存在不同的指令。对于后一种情况则是更常用的，指令

IMUL REG, src

将单倍长度的乘积放到目标单元中，这个目标单元必须是通用寄存器REG。源操作数可以在寄

寄存器或内存中。对于第一种情况, 指令

**IMUL src**

使用EAX寄存器作为目标寄存器, 源操作数可以在寄存器或内存中。双倍长度乘积放在寄存器EDX (高位部分) 和EAX (低位部分) 中。

整数的除法指令格式是

**IDIV src**

源操作数 (src) 是除数。被除数假定是在寄存器EAX中。结果中的商保存在EAX中, 余数保存在EDX中。在指令执行之前, EAX中的被除数必须进行符号扩展, 然后放到EDX中。这是由指令CDQ (convert doubleword to quadword —— 将双倍字长转换成四倍字长) 来完成的, 该指令没有操作数, 因为使用的寄存器假定是EAX和EDX。

浮点数将在第6章中描述, 它具有比整数更大的范围, 主要用于科学计算中。对整个范围内的算术操作都提供了指令。操作数和结果保存在浮点寄存器中, 如图3-37所示。在本指令集中支持单精度 (32位) 和双精度 (64位) 两种格式。

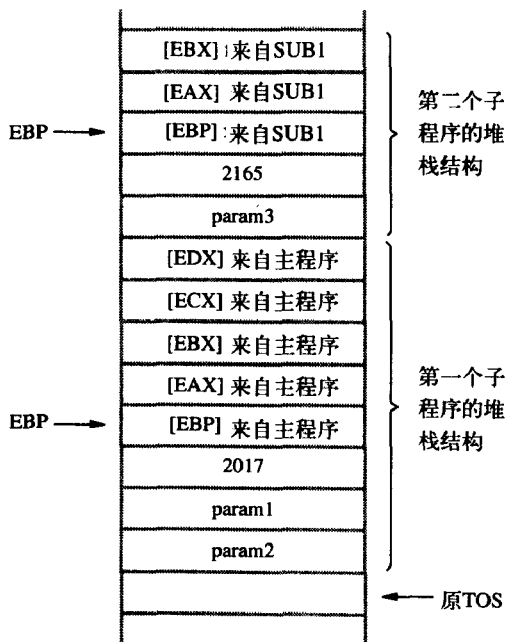


图3-48 图3-47的IA-32堆栈结构

### 3.23.2 多媒体扩展 (MMX) 指令

一个二维图形或视频图像可以用一个包含大量采样图像点的数组来表示。每个点的颜色和亮度称为像素, 可以被编码成一个8位的数据项。这样的数据处理具有两个主要特点。第一个特点是单个像素的操作通常只包括非常简单的算术或逻辑操作, 另一个特点是对于一些实时的应用可能需要非常高的计算速度。而对于采样的音频信号或语音处理 (在一定的时间间隔内对连续的模拟信号进行采样并用一系列带符号数来表示) 来说, 也具有这些特点。

在这两种应用中, 如果单个数据项是字节或16位字, 并可以打包成能并行处理的小组, 那么就可以获得很高的处理效率。IA-32指令系统中有很多将这样的数据打包成64位的四字指令 (一个四字中包含八个字节或四个16位的字)。这些指令称为多媒体扩展 (MMX) 指令。操作数可以在内存或八个浮点寄存器中, 这样, 这些寄存器具有两种服务目的。它们可以保存浮点数或MMX操作数; 当被MMX指令使用时, 这些寄存器作为MM0到MM7来引用。

在内存与MMX寄存器之间提供了传送64位MMX操作数的传送指令。指令

**PADDB MMi, src**

将两个8字节操作数中的相应字节独立相加, 并将八个相加和放到目的寄存器中。源操作数可以在内存或在一个MMX寄存器中, 但是目的地必须是一个MMX寄存器。对于减法和逻辑操作也提供了类似的指令。

在信号处理中的一个公共操作是将输入信号样本的一个短时间序列与称为权的常量相乘, 并将乘积相加产生一个输出信号样本。还有一个将乘法和加法操作合并的MMX指令。它可以对

包含四个16位的信号样本数据项的64位MMX操作数进行操作。

### 3.23.3 向量 (SIMD) 指令

这里提供了一组用来对小组的浮点数进行算术操作的指令。在科学计算中，SIMD（单指令多数据）对于向量和矩阵计算是非常有用的。在Intel术语中，这些指令被称为流SIMD扩展（SSE）指令。它们处理的是由128位构成的操作数，每个操作数由四个浮点数组成。这些操作数可以使用八个128位寄存器来保存（这些寄存器在图3-37中没有给出）。其中，加法和乘法是它们的两种基本操作。它们对128位的复合源和目标操作数中的四对相应的32位操作数进行操作，并将四个独立的结果保存到128位的目标单元中。

## 3.24 实例程序

本节给出2.11节中描述的实例程序的IA-32代码。

### 3.24.1 向量点积程序

图3-49所示的是对保存在内存中、起始地址是AVEC和BVEC的两个数值向量，计算它们点积的IA-32程序。该程序是模仿图2-33中的一般的程序编写的。在图3-49中，用带有变址的基址寻址方式来对每个向量进行连续访问。寄存器EDI当作变址寄存器。由于向量元素假定为双字（4字节）数，所以使用了一个比例因子4。寄存器ECX当作循环计数器，初始值为 $n$ 。这里允许使用LOOP指令（参见3.17节和图3-40b），首先ECX减1，之后如果ECX的内容没有到达0，就条件转移到目的地址LOOPSTART处。两个向量元素的乘积假定是可以放在一个双字中的，乘法指令IMUL明确指定了所需的目标寄存器EDX，正如在3.23节中描述的那样。

184

	LEA	EBP,AVEC	EBP指向向量A
	LEA	EBX,BVEC	EBX指向向量B
	MOV	ECX,N	ECX用作循环计数器
	MOV	EAX,0	EAX累积点积
	MOV	EDI,0	EDI是变址寄存器
LOOPSTART:	MOV	EDX,[EBP+EDI * 4]	计算下一元素的点积
	IMUL	EDX,[EBX+EDI * 4]	
	INC	EDI	增加变址
	ADD	EAX,EDX	加到先前的和中
	LOOP	LOOPSTART	如果没完成就转移返回
	MOV	DOTPORD,EAX	将点积保存到内存中

图3-49 IA-32点积程序

### 3.24.2 字节排序程序

图3-50b中所示的是图2-34b中的字节排序的IA-32代码程序。寄存器EAX中装入地址LIST，用带有变址的基址寻址方式访问列表中当作基址的寄存器。寄存器EDI在外部循环中当作变址 $j$ 的变址寄存器，寄存器ECX作为内部循环的变址 $k$ 的变址寄存器。寄存器DL在遍历每个子表时保存当前最大的字节。在图3-50b与图2-34b的程序中，除了LIST( $k$ )与LIST( $j$ )的交换指令不同

之外，其余的指令都是相同的。在IA-32代码中使用一条单指令XCHG就可以实现这个功能，而在一般的代码中要用到三条指令和一个临时寄存器。

```

for (j = n-1; j > 0; j = j - 1)
{
    for (k = j-1; k >= 0; k = k - 1)
    {
        if (LIST[k] > LIST[j])
        {
            TEMP = LIST[k];
            LIST[k] = LIST[j];
            LIST[j] = TEMP;
        }
    }
}

```

a) C语言的排序程序

	LEA	EAX,LIST	载入列表指针基址寄存器 (EAX),
	MOV	EDI,N	并初始化外部循环变址寄存器 (EDI)
	DEC	EDI	为j = n - 1
OUTER:	MOV	ECX,EDI	初始化内部循环变址寄存器 (ECX)
	DEC	ECX	为k = j - 1
INNER:	MOV	DL,[EAX + EDI]	将LIST(j) 载入到寄存器DL中
	CMP	[EAX + ECX],DL	将LIST(k) 和LIST(j) 进行比较
	JLE	NEXT	如果LIST(k) < LIST(j) 转向下面
			较低的变址项
	XCHG	[EAX + ECX],DL	否则，交换LIST(k) 与LIST(j) 将
			LIST(j) 放到DL中
NEXT:	MOV	[EAX + EDI],DL	
	DEC	ECX	对内部循环变址k减1
	JGE	INNER	重复或终止内部循环
	DEC	EDI	对外部循环变址j减1
	JG	OUTER	重复或终止外部循环

b) 采用IA-32实现的程序

图3-50 采用直接选择排序法的IA-32的字节排序程序

### 3.24.3 链表的插入与删除子程序

图3-51和图3-52中的插入和删除子程序与图2-37与图2-38中的一般程序对应比较，可以看出是非常相近的。采用寄存器来传送参数，寄存器名为RHEAD、RNEWREC、RIDNUM、RCURRENT和RNEXT，这些对于一般程序中的用法都是相同的。这里继续使用了这些名称，而没有使用IA-32中的EAX、EBX等。第六个寄存器RNEWID用来保存要插入的新记录ID号。在图3-51的子程序中，第一条指令将新记录的ID号装入到其中。子程序中的其余部分指令与图2-37中的子程序对应相同。删除一条记录子程序如图3-52所示，也是与图2-38中的子程序逐条指令对应相同的。

正如图2-37和图2-38中的一般程序一样，IA-32的插入子程序假定新记录的ID号与表中的任意记录都不相同，IA-32的删除子程序假定在表中存在有一条与RIDNUM内容相同的ID号记录。

习题3.72和3.73考虑的是如果假设不成立，将如何修改子程序从而给出一个报错信息。

<b>子程序</b>		
INSERTION:	MOV	RNEWID,[RNEWREC]
	CMP	RHEAD,0
	JG	HEAD
	MOV	RHEAD,RNEWREC
HEAD:	RET	
	CMP	RNEWID,[RHEAD]
	JG	SEARCH
	MOV	[RNEWREC+4],RHEAD
SEARCH:	MOV	RHEAD,RNEWREC
	RET	
	MOV	RCURRENT,RHEAD
	MOV	RNEXT,[RCURRENT+4]
LOOPSTART:	CMP	RNEXT,0
	JE	TAIL
	CMP	RNEWID,[RNEXT]
	JL	INSERT
INSERT:	MOV	RCURRENT,RNEXT
	JMP	LOOPSTART
	MOV	[RNEWREC+4],RNEXT
	MOV	[RCURRENT+4],RNEWREC
TAIL:	RET	

图3-51 在链表中插入一条新记录的IA-32子程序

<b>子程序</b>		
DELETION:	CMP	RIDNUM,[RHEAD]
	JG	SEARCH
	MOV	RHEAD,[RHEAD+4]
	RET	
SEARCH:	MOV	RCURRENT,RHEAD
	MOV	RNEXT,[RCURRENT+4]
	CMP	RIDNUM,[RNEXT]
	JE	DELETE
DELETE:	MOV	RCURRENT,RNEXT
	JMP	LOOPSTART
	MOV	RTEMP,[RNEXT+4]
	MOV	[RCURRENT+4],RTEMP
	RET	

图3-52 从链表中删除一条记录的IA-32子程序

### 3.25 结束语

本章讨论了三种不同的指令集体系结构——ARM、Motorola 68000和 Intel IA-32。ARM和68000指令集分别对RISC和CISC设计风格进行了说明。IA-32指令集是CISC设计的特殊扩展。

ARM指令集中的任意指令都编码成一个32位的字。只有数据在处理器寄存器中时才可以对数据进行操作。在处理器寄存器和内存之间使用Load和Store指令来传送操作数。我们阐述了全部指令的条件执行，以及在大部分指令中传送操作数的扩展选项，这可以对普通的程序任务产

生高效的实现效率。

68000是指令可以被编码成可变内存字的指令集的传统例子，这需要根据将被执行的操作数的操作复杂性，以及产生所需内存操作数的有效地址所需信息的情况来定。在个别指令中可以使用寄存器和内存操作数。

Intel IA-32指令集对不同数据类型进行更大范围的操作，它具有更大的灵活性。

## 习题

### 部分I ARM

3.1 假定在ARM计算机中寄存器和内存内容如下：

寄存器R0的内容是1000；

寄存器R1的内容是2000；

寄存器R2的内容是1016；

寄存器R6的内容是20；

寄存器R7的内容是30；

数1, 2, 3, 4, 5和6保存在内存以1000开始的连续地址字中。对于下面的三个短指令块，每次都是以给定的初始值作为起始地址，请说出每个指令块执行后的作用是什么？

- (a) LDR R8, [R0]  
LDR R9, [R0, #4]  
ADD R10, R8, R9
- (b) STR R6, [R1, # - 4]!  
STR R7, [R1, # - 4]!  
LDR R8, [R1], #4  
LDR R9, [R1], #4  
SUB R10, R8, R9
- (c) LDMIA R2!, {R4, R5}  
ADD R4, R4, R5

3.2 下面哪条ARM指令将会使汇编程序指示符产生语法错误信息？为什么？

- (a) ADD R2, R2, R2  
(b) SUB R0, R1, [R2, #4]  
(c) MOV R0, #2 1010101  
(d) MOV R0, #257  
(e) ADD R0, R1, R11, LSL #8

3.3 当使用Load指令将一个字节从内存装入到ARM处理器寄存器中时，其高24位要被清为0（参见3.1.2节）。如果装入的字节使用的是8位带符号的补码表示形式，就必须在它用于算术操作之前，对其所在的寄存器进行符号扩展而变成32位。假设有这样的一个字节已经装入到寄存器R0中，编写一个程序将其进行带符号扩展，成为32位寄存器的长度。（提示：从2.10.2节中描述及图2-30所示的LSL、LSR和ASR中选择适当的移位指令进行移位之后，再用MOV指令将R0的内容传送回R0中。）

- 3.4 编写一个ARM程序，将寄存器R2中的位序进行翻转。例如，如果R2的最初形式是1110...0100，那么在R2中的结果就应该是0010...0111。（提示：使用移位和循环移位操作。）
- 3.5 程序跟踪是在程序执行期间某个特定的寄存器和内存位置处在不同时刻的内容列表。列出在图3-7中程序的BGT指令的前三条指令每次执行之后，寄存器R0、R1和R2中的内容。用表格的形式给出结果，表格以三个寄存器作为列头。用三行列出在BGT指令每次执行完之后寄存器的内容。程序数据在图3-8给出。
- 3.6 编写一个ARM程序，对两个字节列表中的相应字节进行比较，将较大的字节放到第三个列表中。两个列表的起始位置分别是X和Y，较大字节列表的起始位置是LARGER。列表的长度保存在内存单元N处。
- 3.7 一个ARM程序具有如下的字符操作任务：一个具有 $n$ 个字符的字符串保存在内存以STRING为起始地址的连续字节单元处。另一个较短有 $m$ 个字符的字符串保存在以SUBSTRING为起始地址的连续字节单元处。该程序必须在以STRING单元开始的字符串中进行查找，看是否存在一个包含有与SUBSTRING单元保存的字符串相匹配的连续子串。长度参数 $n$ 和 $m$ （其中 $n > m$ ）分别保存在内存中的N和M处。如果找到了一个匹配的子串，将它的第一个字节地址保存在寄存器R0中；否则，R0的内容将被清0。该程序不必判断子串是否重复出现。这里只需要第一个匹配子串的地址。
- 3.8 编写一个ARM程序，生成一个长度为 $n$ 的Fibonacci数列。数列中的前两个数为0和1，每个数等于前面两个数之和。例如， $n = 8$ 时的数列为

0, 1, 1, 2, 3, 5, 8, 13

程序中应该在以MEMLOC开始的连续内存字中保存。假设 $n$ 值保存在单元N处。

- 3.9 编写一个ARM程序，将一个单词（文本）的全部小写字母转换成大写字母。构成该单词的ASCII字符保存在内存连续的字节中，它的起始位置为WORD，以一个空格为结束（参见附录E中的ASCII表）。
- 3.10 对图2-14中的成绩表稍做修改，每个学生有 $j$ 门测验成绩。假设有 $n$ 个学生。编写一个ARM程序用于计算每门测验的总分数并将这些总分数保存到内存地址为SUM, SUM+4, SUM+8, ...的字中。测验的数目 $j$ 要大于处理器中寄存器的数量，所以在图2-15中给出的用于3门测验情况的程序类型是不能使用的。建议使用在2.5.3节中介绍的两个嵌套循环。内部循环用于累计某个特定的测验总和，外部循环用于控制测验的次数 $j$ 。假设 $j$ 保存在内存中的单元J处，该单元在单元N的前面。
- 3.11 考虑一个数值数组A ( $i, j$ )，它的行变址从 $i = 0$ 到 $n-1$ ，列变址从 $j = 0$ 到 $m-1$ 。该数组在ARM计算机上采用按行存储，每行元素占用 $m$ 个连续的字。编写一个用于将列 $x$ 和列 $y$ 按对应的元素相加，并将相加的和保存在列 $y$ 中的ARM子程序。变址 $x$ 和 $y$ 是通过寄存器R1和R2传送给子程序的。参数 $n$ 和 $m$ 通过寄存器R3和R4传送给子程序，元素A (0, 0)的地址用寄存器R0来传送。可以使用表3-1中所列出的寻址方式。
- 3.12 编写一个ARM程序，从键盘读取 $n$ 个字符，在读取这些字符时将它们放入用户使用的堆栈中，之后将这些字符在显示器上显示出来。使用寄存器R6作为堆栈指针。字符个数 $n$ 保存在内存中的N处。
- 3.13 假设在图3-9的程序中的一条指令从取出到执行所用的平均时间是20纳秒。如果每秒钟键盘的按键次数是10，那么每输入一个字符时，BEQ READ指令大概执行多少次？这里假设每



190

个字符的显示时间要远远小于键盘上连续输入两个字符的时间。

- 3.14 在图3-9中的ARM程序中,“in-line”代码完成的功能是读取一行字符并将它们显示出来。重新编写该程序,采用主程序调用名为GETCHAR的子程序来读取一个字符,然后再调用另一个名为PUTCHAR的子程序来显示单个字符。地址INSTATUS通过寄存器R1来传送给GETCHAR,主程序利用寄存器R3来获取所需要的字符。地址OUTSTATUS和要显示的字符分别通过寄存器R2和R3传送给PUTCHAR。子程序中使用的其他寄存器必须通过堆栈来保存和恢复,堆栈的指针是寄存器R13。在主程序中完成字符在内存中的排序以及查找行尾字符CR的工作。
- 3.15 采用堆栈来传送参数,解答习题3.14。
- 3.16 编写一个从键盘接收三个十进制数的ARM程序。每个数用ASCII码(参见附录E)表示。假设这三个十进制数表示的范围在0到999的十进制整数之中,将该整数转换成一个二进制数表示形式,并首先接受数的高位部分。为了简便起见,在内存中保存两张字表,每张表中有10项内容。第一张表的起始位置是TENS,其内容是十进制数0, 10, 20, ..., 90的二进制表示形式。第二张表的起始位置是HUNDREDS,其内容是十进制数0, 100, 200, ..., 900的二进制表示形式。
- 3.17 对习题3.16中的十进制到二进制转换程序使用两个嵌套的子程序来实现。调用第一个子程序的主程序通过将两个参数压入堆栈(指针寄存器是R13)来传送参数。第一个参数是用来保存输入的十进制数字字符的一个3字节内存缓冲区的地址。第二个参数是转换后的二进制数的保存单元的地址。第一个子程序从键盘读取三个字符,然后调用第二个子程序进行转换。该子程序所需参数是通过处理器中的寄存器进行传送的。两个子程序必须在堆栈中保存它们使用的任何寄存器的内容。
- (a) 为ARM处理器编写这两个子程序。
- (b) 给出第二个子程序调用指令执行完成时堆栈的内容。
- 3.18 考虑习题2.18中描述的队列结构。编写一个在处理器寄存器和队列之间传送数据的ARM APPEND和REMOVE 程序。要注意检查和更新队列的状态以及每次试图执行一个操作的指针情况。
- 3.19 利用习题3.5中描述的结果表示格式,对图3-15b中的字节排序程序编写一个跟踪程序。在程序中最后一条指令每次执行之后,给出寄存器R0、R2和R3的内容和位于LIST, LIST+1, ..., LIST + 4的5字节列表的内容。假设LIST=1000,列表的初始值是120、13、106、45及67,其中[LIST]=120。
- 191
- 3.20 用一个子程序重写图3-15b中的字节排序程序,对32位的正整数列表进行排序。调用程序将列表地址传送给子程序。在该位置处的第一个32位数是列表项的数目,之后紧跟着的就是要排序的数字。
- 3.21 考虑图3-15b的字节排序程序。在每次遍历子列表期间,即从LIST(j)到LIST(0),只要是LIST(k) > LIST(j)就将表项交换。另一种策略是保持子列表中最大值的地址,在子列表查找的末尾处进行一次对换。利用这种方法重写程序。这种方法有何优点?
- 3.22 假设图2-14所示的学生测试成绩列表保存在内存的一个链表(如图2-36所示)中。编写一个与图2-15程序具有相同功能的ARM程序。头记录保存在内存的1000处。
- 3.23 图3-16中的链表插入子程序并没有检查是否在链表中存在与新记录相同的记录。如果在链

表中存在这样的记录，执行子程序会如何呢？修改这个子程序，使之在找到匹配的记录时将该记录的地址存入寄存器R10中并返回，如果插入成功返回0。

- 3.24 在图3-17中的链表删除子程序中，假设列表中存在有寄存器RIDNUM所包含的ID记录。如果这个ID的记录在链表中不存在，执行子程序将会如何？修改这个子程序，如果删除成功，用RIDNUM返回0；如果该记录在链表中不存在，就保持RIDNUM不变。

## 部分II 68000

- 3.25 考虑以下68000处理器的状态：

寄存器D0的内容是\$1000；

寄存器A0的内容是\$2000；

寄存器A1的内容是\$1000；

内存单元\$1000处保存一个长字\$2000；

内存单元\$2000处保存一个长字\$3000；

如果下面的三条指令每次都是以这个初始状态开始，那么每条指令执行完的结果如何？每条指令要占用多少字节？每条指令读取和执行的过程中需要访问多少次内存？

- (a) ADD.L D0, (A0)
- (b) ADD.L (A1,D0), D0
- (c) ADD.L #\$2000, (A0)

- 3.26 从下面的68000指令中找出语法错误：

- (a) ADDX - (A2), D3
- (b) LSR.L #9, D2
- (c) MOVE.B 520(A2, D2)
- (d) SUBA.L 12(A2, PC), A0
- (e) CMP.B #254, \$12(A2, D1.B)

- 3.27 程序跟踪是在一个程序执行期间某个特定的寄存器和内存位置处在不同时刻的内容列表。在ADD.W指令每次执行完以及图3-25程序中最后的MOVE.L指令执行完之后，列出寄存器D0、D1和A2以及内存单元N、NUM1和SUM中的内容。以表格的形式给出结果，表格中以寄存器和内存单元作为列的头部。使用六行将每条指令执行完之后的寄存器和内存的内容列出来。假定有下面的初始值：[SUM]=0，[N]=5，NUM1=2400，五个数分别是83、45、156、-250和100。

- 3.28 考虑下面的68000程序：

```

MOVEA.L    MEM1, A0
MOVEA.L    MEM2, A2
ADDA.L     A0, A1
MOVEA.L    A0, A2
MOVE.B     (A0) +, D0
LOOP CMP.B  (A0) +, D0
BLE        NXT
LEA        -1(A0), A2
    
```

```

                MOVE.B    (A2), D0
NXT             CMPA.L    A0, A1
                BGT       LOOP
                MOVE.L    A2, DESIRED

```

- (a) 这个程序的功能是什么？  
 (b) 将这个程序保存到内存中需要多少16位的字？  
 (c) 给出访问内存所需次数的表达式。表达式可以是  $T = a + bn + cm$  的形式，其中  $n$  是循环要执行的次数， $m$  是不执行到 NXT 处转移的次数， $a$ 、 $b$  和  $c$  是常量。

3.29 考虑图P3-1中的两个68000程序。

193

- (a) 这两个程序在 RSLT 处的值相同吗？  
 (b) 它们完成的任务是什么？  
 (c) 在内存中保存每个程序各需多少字节？  
 (d) 哪一个程序对内存的访问次数更多？  
 (e) 这两个程序的优点和缺点是什么？

程序1			程序2		
	CLR.L	D0		MOVE.W	#\$FFFF,D0
	MOVEA.L	#LIST,A0		MOVEA.L	#LIST,A0
LOOP	MOVE.W	(A0)+,D1	LOOP	LSL.W	(A0)+
	BGE	LOOP		BCC	LOOP
	ADDQ.L	#1,D0		LSL.W	#1,D0
	CMPI	#17,D0		BCS	LOOP
	BLT	LOOP		MOVE.W	-2(A0),RSLT
	MOVE.W	-2(A0),RSLT			

图P3-1 习题3.29的两个68000程序

- 3.30 编写一个68000程序，对两个字节列表中相应的字节进行比较并将较大字节的列表放到第三个表中。两个列表的起始位置分别是X和Y，较大字节列表的起始位置是LARGER。列表的长度保存在内存中N单元处。
- 3.31 一个68000程序具有如下的字符操作任务：一个具有  $n$  个字符的字符串保存在内存中以 STRING 为起始地址的连续字节单元处。另一个较短的有  $m$  个字符的字符串保存在以 SUBSTRING 为起始地址的连续字节单元处。该程序必须从 STRING 单元处保存的字符串开始查找，看是否存在一个与 SUBSTRING 单元保存的字符串相匹配的连续子串。长度参数  $n$  和  $m$ （其中  $n > m$ ）分别保存在内存中的 N 和 M 单元处。如果找到了一个匹配的子串，将它的第一个字节的地址保存在寄存器 R0 中；否则，R0 的内容将被清0。该程序不必判断子串是否重复出现。这里只需要第一个匹配子串的地址。
- 3.32 编写一个68000程序，生成一个长度为  $n$  的 Fibonacci 数列。数列中的前两个数为0和1，每个数等于前面两个数之和。例如， $n = 8$  时的数列为

0, 1, 1, 2, 3, 5, 8, 13

程序应该在以 MEMLOC 开始的连续内存字中保存。假设  $n$  值保存在单元 N 处。程序中可以处理的最大  $n$  值是多少？

- 3.33 编写一个68000程序，将一个单词（文本）的全部小写字母转换成大写字母。构成该单词的ASCII字符保存在内存连续的字节中，它的起始位置为WORD，以一个空格为最后结束（参见附录E中的ASCII表）。 194
- 3.34 对图2-14中的成绩表修改一下，使每个学生有  $j$  门测验成绩。假设有  $n$  个学生。编写一个68000程序用于计算每门测验的总分数并将这些总分数保存到内存地址为SUM, SUM+4, SUM+8, ...的字中。测验的数目  $j$  要大于处理器中寄存器的数量，所以在图2-15中给出的用于3门测验情况的程序类型是不能使用的。建议使用在2.5.3节中介绍的两个嵌套循环。内部循环用于累计某个特定的测验总和，外部循环用于控制测验的次数  $j$ 。假设  $j$  保存在内存中单元J处，该单元在单元N的前面。
- 3.35 编写一个68000程序，从键盘读取  $n$  个字符，在读取这些字符时将它们放入用户使用的堆栈中，之后将这些字符在显示器上显示出来。使用寄存器A0作为堆栈指针。字符个数  $n$  保存在内存的单元N处。
- 3.36 假设在图3-27程序中的一条指令从取出到执行所用的平均时间是20纳秒。如果每秒钟键盘的按键次数是10，那么每输入一个字符时，BEQ READ指令大概执行多少次？假设每个字符的显示时间要远远小于键盘上连续输入两个字符的时间。
- 3.37 在图3-27的68000程序中，“in-line”代码的功能读取一行字符并将它们显示出来。重新编写该程序，采用主程序调用名为GETCHAR的子程序来读取一个字符，然后再调用另一个名为PUTCHAR的子程序来显示单个字符。地址INSTATUS和DATAIN通过寄存器A0和A1来传送给GETCHAR，主程序利用寄存器D0来获取所要的字符。地址OUTSTATUS与DATAOUT和要显示的字符分别通过寄存器A2、A3和D0传送给PUTCHAR。子程序中使用的其他寄存器必须由子程序通过堆栈进行保存和恢复，堆栈的指针是寄存器A7。在主程序中完成字符在内存中的排序以及查找行尾字符CR的工作。
- 3.38 采用堆栈传送参数的方法重新回答3.37的问题。
- 3.39 考虑习题2.18中描述的队列结构。编写一个在处理器寄存器和队列之间传送数据的68000 APPEND和REMOVE 程序。要注意检查和更新队列的状态以及每次试图执行一个操作的指针情况。
- 3.40 编写一个从键盘接收三个十进制数的68000程序。每个数用ASCII码（参见附录E）表示。假设这三个十进制数表示一个范围在0到999的十进制整数，将该整数转换成一个二进制形式表示的数。首先接受数的高位部分。为了简便起见，在内存中保存两张字表。每张表中有10项内容。第一张表的起始位置是TENS，其内容是十进制数0, 10, 20, ..., 90的二进制表示形式。第二张表的起始位置是HUNDREDS，其内容是十进制数0, 100, 200, ..., 900的二进制表示形式。 195
- 3.41 对习题3.40中的十进制到二进制转换程序使用两个嵌套子程序来实现。调用第一个子程序的主程序通过将两个参数压入堆栈（指针寄存器是R13）来传送参数。第一个参数是用来保存输入的十进制数字字符的一个3字节内存缓冲区的地址。第二个参数是转换后的二进制值保存位置的地址。第一个子程序从键盘读取三个字符，之后调用第二个子程序来进行转换。该子程序所需的参数是通过处理器寄存器来传送的。两个子程序必须在堆栈中保存它们使用的任何寄存器的内容。
- (a) 为68000处理器编写两个子程序。

(b) 给出第二个子程序调用指令执行完时处理器堆栈中的内容。

- 3.42 考虑一个16位的数组A ( $i, j$ )，它的行变址从 $i = 0$ 到 $n - 1$ ，列变址从 $j = 0$ 到 $m - 1$ 。该数组在68000计算机上采用按行存储，每行元素占用 $m$ 个连续的字。编写一个用于将列 $x$ 和列 $y$ 按对应的元素相加，并将相加的和保存在列 $y$ 中的68000子程序。变址 $x$ 和 $y$ 是通过寄存器D1和D2传送给子程序的。参数 $n$ 和 $m$ 通过寄存器D3和D4传送给子程序，元素A (0,0)的地址是用寄存器A0来传送。可以使用在表3-2中所列出的寻址方式。
- 3.43 编写一个68000程序，将寄存器D2中的位序进行翻转。例如，如果D2的最初形式是1110...0100，那么在D2中的结果就应该是0010...0111。(提示：使用移位和循环移位操作。)
- 3.44 将图3-32中的程序保存到内存中需要多少字节？在程序的执行期间，需要进行多少次内存访问？
- 3.45 利用习题3.27中描述的结果表示格式，对图3-34b中的字节排序程序编写一个跟踪程序。在程序中最后一条指令每次执行之后，给出寄存器R0、R2和R3的内容和位于LIST, LIST+1,..., LIST+4的5字节列表中的内容。假设LIST=1000，列表的初始值是120、13、106、45及67，其中[LIST]=120。
- 3.46 用一个子程序重写图3-34b中的字节排序程序，对16位的正整数列表进行排序。调用程序将列表地址传送给子程序。在该位置处的第一个16位数是列表项的个数，之后紧跟着的就是要排序的数字。
- 3.47 考虑图3-34b的字节排序程序。在每次遍历子列表期间，即从LIST ( $j$ ) 到LIST (0)，只要是LIST ( $k$ ) > LIST ( $j$ ) 就将表项交换。另一种策略是保持子列表中最大值的地址，在子列表查找的末尾处进行一次对换。利用这种方法重写程序。这种方法有何优点？
- 196 3.48 假设图2-14所示的学生测试成绩列表保存在内存的一个链表(如图2-36所示)中。编写一个与图2-15程序具有相同功能的68000程序。头记录保存在内存的1000处。假定全部的表项都采用长字。
- 3.49 图3-35中的链表插入子程序并没有检查是否在链表中存在与新记录相同的记录。如果在链表中存在这样的记录，执行子程序将如何做？修改这个子程序，使之在找到匹配的记录时将该记录的地址存入寄存器A6中并返回，如果插入成功则返回0。
- 3.50 在图3-36的链表删除子程序中，假设列表中不存在有寄存器RIDNUM所包含的ID记录。如果这个ID记录在链表中不存在，执行子程序将如何？修改这个子程序，如果删除成功，用RIDNUM返回0；如果该记录在链表中不存在，就保持RIDNUM不变。

### 部分III Intel IA-32

- 3.51 假设IA-32计算机中的寄存器和内存的内容如下：

寄存器EBX的内容是1000；

寄存器ESI的内容是2；

数值1, 2, 3, 4, 5和6保存在内存从1000开始的连续双字单元处；

地址标号LOC代表地址1008；

如果下面的三条指令每次都是以这种初始状态开始的，那么每条指令执行完的结果如何？

每条指令要占用多少个字节?

- (a) MOV EAX, 10  
ADD EAX, [EBX + ESI\*4 + 8]  
(b) PUSH 20  
PUSH 30  
POP EAX  
POP EBX  
SUB EAX, EBX  
(c) LEA EAX, LOC  
MOV EBX, [EAX]

3.52 下面的IA-32指令中哪个会使汇编程序产生语法错误信息? 为什么?

- (a) ADD EAX, EAX  
(b) ADD [EAX], [EBX+4]  
(c) SUB EAX, [EBX+ESI\*4 + 20]  
(d) SUB EAX, [EBX+ESI\*10]  
(e) ADD EAX, - 31728542  
(f) MOV 20, EAX  
(g) MOV EAX, [EBP+ESP\*4]

197

3.53 程序跟踪是在一个程序执行期间对某个特定的寄存器和内存单元在不同时刻内容的列表。列出在图3-40b程序的循环指令的前三条指令每次执行之后, 寄存器EAX、EBX和ECX中的内容。用表格的形式给出结果, 表格以三个寄存器作为列头。用三行列出在循环指令每次执行完之后寄存器中的内容。程序数据在图3-42给出。

3.54 编写一个IA-32程序, 对两个字节列表中的相应字节进行比较并将较大字节的列表放到第三个表中。两个列表的起始位置分别是X和Y, 较大字节列表的起始位置是LARGER。列表的长度保存在内存单元N处。

3.55 一个IA-32程序具有如下的字符操作任务: 一个具有 $n$ 个字符的字符串保存在内存以STRING为起始地址的连续字节单元处。另一个较短的有 $m$ 个字符的字符串保存在以SUBSTRING为起始地址的连续字节单元处。该程序必须在以STRING单元开始的字符串中进行查找, 查看是否存在一个与SUBSTRING处保存的字符串相匹配的连续子串。长度参数 $n$ 和 $m$  (其中 $n > m$ ) 分别保存在内存中的N和M单元处。如果找到了一个匹配的子串, 将它的第一个字节地址保存在寄存器R0中; 否则, R0的内容将被清0。该程序不必判断子串是否重复出现。这里只需要第一个匹配子串的地址。

3.56 编写一个IA-32程序, 生成一个长度为 $n$ 的Fibonacci数列。数列中的前两个数为0和1, 每个数等于前面两个数之和。例如,  $n = 8$ 时的数列为

0, 1, 1, 2, 3, 5, 8, 13

程序应该保存在以MEMLOC开始的连续内存字中。假设 $n$ 值保存在单元N处。

3.57 编写一个IA-32程序, 将一个单词 (文本) 的全部小写字母转换成大写字母。构成该单词的ASCII字符保存在内存连续的字节中, 它的起始位置为WORD, 以一个空格为最后结束。

(参见附录E中的ASCII表。)

- 198 3.58 对图2-14中的成绩表做一些修改,使每个学生有 $j$ 门测验成绩。假设有 $n$ 个学生。编写一个IA-32程序用于计算每门测验的总分数并将这些总分数保存到内存地址SUM, SUM+4, SUM+8, ...的字中。测验的数目 $j$ 要大于处理器中寄存器的数量,所以在图2-15中给出的用于3门测验情况的程序类型是不能使用的。建议使用在2.5.3节中介绍的两个嵌套循环。内部循环用于累计某个特定的测验总和,外部循环用于控制测验次数 $j$ 。假设 $j$ 保存在内存单元J处,该单元在单元N的前面。
- 3.59 编写一个IA-32程序,将寄存器EAX中的位序进行翻转。例如,如果EAX的最初形式是1110...0100,那么在EAX中的结果就应该是0010...0111。(提示:使用移位和循环移位操作。)
- 3.60 考虑习题2.18中描述的队列结构。编写一个在处理器寄存器和队列之间传送数据的IA-32 APPEND和REMOVE 程序。要注意检查和更新队列的状态以及每次试图执行一个操作的指针情况。
- 3.61 编写一个IA-32程序,从键盘读取 $n$ 个字符,在读取这些字符时将它们放入用户使用的堆栈中,之后将这些字符在显示器上显示出来。使用寄存器EBX作为堆栈指针。字符个数 $n$ 保存在内存单元N处。
- 3.62 假设图3-44程序中的一条指令从取出到执行所用的平均时间是20纳秒。如果每秒钟键盘的按键次数是10,那么每输入一个字符时,JNC READ指令大概执行多少次?假设每个字符的显示时间要远远小于键盘上连续输入两个字符的时间。
- 3.63 在图3-44中的IA-32程序中,“in-line”代码的功能是读取一行字符并将它们显示出来。重新编写该程序,采用主程序调用名为GETCHAR的子程序来读取一个字符,然后再调用另一个名为PUTCHAR的子程序来显示单个字符。地址INSTATUS和DATAIN通过寄存器EBX和EDX来传送给GETCHAR,主程序利用寄存器AL来获取所要的字符。地址OUTSTATUS与DATAOUT和要显示的字符分别通过寄存器ESI、EDI和AL传送给PUTCHAR。子程序中使用的其他寄存器必须由子程序通过堆栈来保存和恢复,堆栈的指针是寄存器ESP。在主程序中完成对字符在内存中的排序以及查找行尾字符CR的工作。
- 3.64 采用堆栈传送参数的方法重新回答3.63的问题。
- 3.65 编写一个从键盘接收三个十进制数的IA-32程序。每个数用ASCII码(参见附录E)表示。假设这三个十进制数表示一个范围在0到999的十进制整数,将该整数转换成一个二进制形式表示的数。首先接受数的高位部分。为了简便起见,在内存中保存两张字表。每张表中有10项内容。第一张表的起始位置是TENS,其内容是十进制数0, 10, 20, ..., 90的二进制表示形式。第二张表的起始位置是HUNDREDS,其内容是十进制数0, 100, 200, ..., 900的二进制表示形式。
- 199 3.66 对习题3.65中的十进制到二进制转换程序使用两个嵌套子程序来实现。调用第一个子程序的主程序通过将两个参数压入堆栈(指针寄存器是R13)来传送参数。第一个参数用来保存输入的十进制数字字符的一个3字节内存缓冲区的地址。第二个参数是转换后的二进制值的保存单元地址。第一个子程序从键盘读取三个字符,之后调用第二个子程序进行转换。该子程序所需的参数是通过处理器寄存器来传送的。两个子程序必须在堆栈中保存它们使用的任何寄存器的内容。

- (a) 为IA-32处理器编写两个子程序。
- (b) 给出第二个子程序调用指令执行完时的处理器堆栈中的内容。
- 3.67 考虑一个数值数组 $A(i, j)$ ，它的行变址从 $i = 0$ 到 $n - 1$ ，列变址从 $j = 0$ 到 $m - 1$ 。该数组在IA-32计算机上采用按行存储，每行元素占用 $m$ 个连续的字。编写一个用于将列 $x$ 和列 $y$ 按对应的元素相加，并将相加的和保存在列 $y$ 中的IA-32子程序。变址 $x$ 和 $y$ 是通过寄存器ESI和EDI传送给子程序的。参数 $n$ 和 $m$ 通过寄存器EAX和EBX传送给子程序，元素 $A(0, 0)$ 的地址是用寄存器EDX进行传送的。可以使用在表3-3中所列出的寻址方式。
- 3.68 利用习题3.53中描述的结果表示格式，对图3-50b中的字节排序程序编写一个跟踪程序。在程序中每当最后一条指令执行之后，给出寄存器EDI、ECX和DL中的内容和位于LIST, LIST+1, ..., LIST+4处的5字节列表内容。假设LIST=1000，列表的初始值是120、13、106、45及67，其中[LIST]=120。
- 3.69 用一个子程序重写图3-50b中的字节排序程序，对32位的正整数列表进行排序。调用程序将列表地址传送给子程序。在该单元处的第一个32位数是列表项的个数，之后紧跟着的就是要排序的数字。
- 3.70 考虑图3-50b的字节排序程序。在每次遍历子列表期间，即从LIST( $j$ )到LIST(0)，只要是LIST( $k$ ) > LIST( $j$ )就将表项交换。另一种策略是保持子列表中最大值的地址，在子列表查找的末尾处进行一次对换。利用这种方法重写程序。这种方法有何优点？
- 3.71 假设图2-14所示的学生测试成绩列表保存在内存的一个链表（如图2-36所示）中。编写一个与图2-15程序具有相同功能的IA-32程序。头记录保存在内存的1000处。假定全部表项都采用长字形式。
- 3.72 图3-51中的链表插入子程序并没有检查是否在链表中存在与新记录相同的记录。如果在链表中存在这样的记录，执行子程序将如何做呢？修改这个子程序，使之在找到匹配的记录时将该记录的地址存入寄存器EDX中并返回，如果插入成功则返回0。
- 3.73 在图3-52的链表删除子程序中，假设列表中存在有寄存器RIDNUM所包含的ID记录。如果这个ID的记录在链表中不存在，执行子程序将如何做？修改这个子程序，如果删除成功，用RIDNUM返回0；如果该记录在链表中不存在，就保持RIDNUM不变。

200

## 参考文献

1. D. Jaggard, "ARM Architecture and Systems," *IEEE Micro*, 17(4): 9-11, July/August 1997.
2. S. Furber, *ARM System-on-Chip Architecture*, Addison-Wesley, Harlow, England, 2000.
3. A. Clements, *The Principles of Computer Hardware*, 3rd ed., Oxford University Press, New York, 2000.
4. A. van Someren and C. Attack, *The ARM RISC Chip — A Programmer's Guide*, Addison-Wesley, Wokingham, England, 1994.
5. <http://www.arm.com>
6. <http://www.motorola.com>
7. D. Tabak, *Advanced Microprocessors*, McGraw-Hill, New York, 1991.



8. <http://www.intel.com>
9. B.B. Brey, *The Intel Microprocessors*, 5th ed., Prentice-Hall, Upper Saddle River, New Jersey, 2000.
10. S.P. Dandamudi, *Introduction to Assembly Language Programming — From 8086 to Pentium Processors*, Springer-Verlag, New York, 1998.

## 输入输出组织结构

## 本章目标

在本章中你将学习以下内容:

- 如何使用轮询方式执行程序来控制I/O
- 中断的概念以及支持中断的硬件和软件
- 高速设备中直接存储器访问的I/O机制
- 通过同步和异步总线的数据传送
- I/O接口电路的设计
- 在特定的PCI、SCSI和USB总线中的商用总线标准

203

计算机的一个基本特性是具有与其他设备交换信息的能力。这一通信能力使操作员可以完成很多操作,例如用键盘和显示屏幕来处理文本和图形。实际中,我们广泛地使用计算机通过Internet与其他计算机通信并访问全球信息。此外,在其他应用中,计算机虽然不是很直观,但仍有着同等重要的作用。它们是家用电器、制造设备、运输系统、银行和销售点终端的集成部分。在这些应用中,计算机的输入可能来自传感器开关、数字照相机、麦克风或火警报警器;输出可能是发送给扬声器的声音信号或用来改变发动机速度、打开阀门或使机器人按指定方式传送的数字编码指令。简而言之,计算机总的目标是在不同环境下与多种设备交换信息。

在本章中,我们将详细分析执行I/O操作的各种不同方法。首先讨论程序员遇到的一些问题,然后讨论与总线、I/O接口有关的硬件细节,并介绍一些通用的总线标准。

## 4.1 访问I/O设备

最简单的就是使用单总线方案将I/O设备连接到计算机上,如图4-1所示。这根总线使所有与它相连的设备可以交换信息。通常由三种传输线组成,分别用来传输地址、数据和控制信号。每台设备都分配一个惟一的地址空间。当处理器将特定的地址放置到地址线上时,识别出这个地址的设备就响应控制线上的命令。处理器进行读或写请求,请求的数据由数据线传输。就像我们在2.7节中提到的,当I/O设备与存储器共享同一个地址空间时,这种方式称为存储器

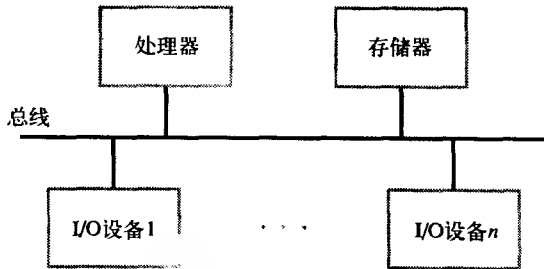


图4-1 单总线结构

204

映射I/O (memory-mapped I/O)。

具有存储器映射I/O时,任何可以访问存储器的机器指令也都可以用来与I/O设备进行数据传送。比如,如果DATAIN是键盘输入缓冲区的地址,指令

Move DATAIN, R0

将从DATAIN中读取数据并存储到寄存器R0中。类似地,指令

Move R0, DATAOUT

将寄存器R0的内容发送到DATAOUT, DATAOUT可能是显示单元或打印机的输出数据缓冲区。

大部分计算机系统采用的是存储器映射I/O。但也有些处理器用专门的输入输出指令来执行I/O传输。例如,第3章中描述的Intel系列处理器就有专门的I/O指令和分配给I/O设备的独立的16位地址空间。当构建一个基于这些处理器的计算机系统时,设计者可以有两种方法连接I/O设备:使用专门的I/O地址空间或简单地将它们作为存储器地址空间的一部分。后一种方法目前最常用,因为它可以使软件简化。独立I/O地址空间的一个优点是:I/O设备需要处理的地址线少。但需要注意的是,独立的地址空间并不意味着I/O地址线与存储器地址线在物理上是分开的。总线上有一个专门的信号用来表示请求的读/写传输是否为I/O操作。如果该信号被置位,则存储器忽略此次传输的请求。I/O设备检查地址线上的低位,然后决定是否应该响应。

图4-2说明了将I/O设备连接到总线上所需要的硬件。地址译码器可以使设备在地址线上识别出自己的地址。数据寄存器中保存正在发送给处理器或者来自处理器的数据。状态寄存器包含有关I/O设备的操作信息。数据和状态寄存器都连接到数据线上并分配惟一的地址。控制电路用来协调I/O传输。地址译码器、数据和状态寄存器以及控制电路构成了设备的接口电路。

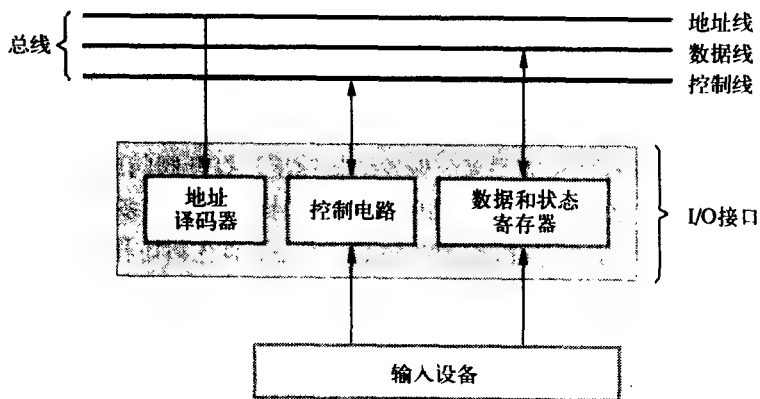


图4-2 输入设备的I/O接口

I/O设备的速度与处理器的速度相差非常大。当操作者从键盘输入数据时,在连续的两次输入之间处理器就可以处理上百万条指令。只有当键盘接口的输入缓冲区中存在有效字符时,键盘读取指令才能被执行。同时还必须确保每个字符只被读取一次。

在2.7节中我们介绍了输入输出操作的一些基本概念。在输入输出设备中,如键盘,状态标志“SIN”作为状态寄存器的一部分包含在接口电路中。当在键盘上按下某个字符时,SIN被置1,当处理器读取该字符后它被清0。因此,软件可以通过检查SIN标志来保证每次读取的都是有效数据。通常用一个重复读取状态寄存器并检查SIN状态的循环程序来执行这一功能。当SIN等于1时,程序读取输入数据寄存器。对于输出操作也可以用相似的程序控制,使用输出状态

标志SOUT。

**例4.1** 复习一下基本概念，我们来看一个计算机系统中简单的I/O操作的例子，该操作涉及一个键盘和一台显示器。图4-3中所示的4个寄存器用于数据传输操作。寄存器STATUS包括两个控制标志SIN和SOUT，分别为键盘和显示器提供状态信息。该寄存器中的另外两个标志KIRQ和DIRQ与中断协同工作。这两个标志与CONTROL寄存器中的KEN和DEN位将在4.2节中讨论。从键盘输入的数据存储在DATAIN寄存器中，发送给显示器的数据存储在DATAOUT寄存器中。

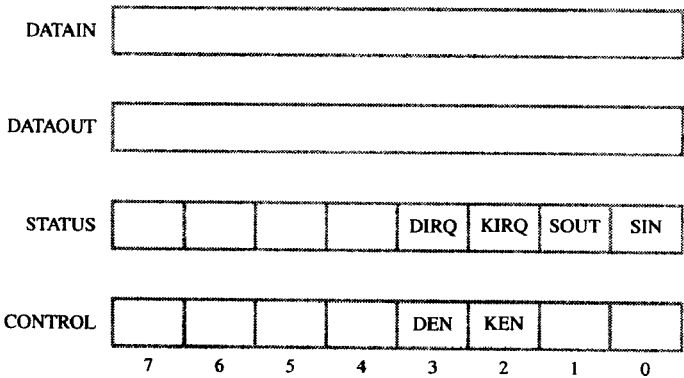


图4-3 键盘和显示器接口中的寄存器

206

图4-4所示的程序与图2-20所示的程序类似。该程序从键盘读入一行字符并存储到从LINE位置开始的存储器缓冲区中，然后调用子程序PROCESS对输入行进行处理。每读取一个字符就将它回显（echoed back）到显示器上。寄存器R0用作存储器缓冲区的指针。由于R0中的内容采用自动递增寻址方式更新，因此连续的字符将存储在连续的存储器区域中。

	Move	#LINE,R0	初始化存储器指针
WAITK	TestBit	#0,STATUS	测试SIN
	Branch=0	WAITK	等待输入一个字符
	Move	DATAIN,R1	读取字符
WAITD	TestBit	#1,STATUS	测试SOUT
	Branch=0	WAITD	等待显示器就绪
	Move	R1,DATAOUT	发送字符到显示器
	Move	R1,(R0)+	存储字符并将指针增加1
	Compare	#\$0D,R1	检查是否为回车符
	Branch ≠ 0	WAITK	如果不是，得到另一字符
	Move	#\$0A,DATAOUT	否则，发送换行符
	Call	PROCESS	调用子程序处理输入行

图4-4 从键盘读入一行字符存储到存储器缓冲区，并在显示器上回显的程序

对每个字符，都要检查是否为回车符（CR），回车符的ASCII码值为0D（16进制）。如果是，那么就发送一个换行符（ASCII码值为0A）将显示器上的光标下移一行并调用子程序PROCESS。否则，程序循环等待下一个输入字符。

该程序阐明了程序控制I/O的工作方式，这里处理器通过不断地检查状态标志来实现与输入输出设备的同步，这称为处理器轮询设备。此外，还有其他两种通用的I/O操作方式：中断和直

接存储器访问。在中断方式中,当I/O设备准备好传输操作时就在总线上发送一个专门的信号,由此实现同步。直接存储器访问是一种用于高速I/O设备的技术。在这种方式中,设备接口与主存之间直接传输数据,不再需要处理器的干预。在接下来的三节中我们将分别讲述这三种方式,然后分析涉及到的硬件,包括处理器和I/O设备接口。

[207]

## 4.2 中断

在图4-4的例子中,当程序进入等待循环后,不停地重复测试设备状态。这期间,处理器不进行任何有效的计算。但在很多情况下,处理器在等待I/O设备就绪期间可以执行其他的任务。为此,我们安排I/O设备在准备就绪时主动通知处理器,这可以通过发送硬件中断信号给处理器来实现。在这种方式下,至少需要一条专门的中断请求控制线。既然处理器不再需要连续检测外部设备,便可以利用等待时间来执行其他任务。实际上,通过使用中断可以很理想地消除这些等待时间。

**例4.2** 让我们来看一项任务,它首先需要进行一些计算,然后将结果在行式打印机上打印出来。之后是更多的计算和输出,如此继续。假设程序包括两个子程序: COMPUTE和PRINT, COMPUTE产生 $n$ 行输出的集合,这些输出行将由PRINT打印出来。

这一任务可以通过先执行COMPUTE程序,然后执行PRINT程序,如此重复来完成。由于打印机每次只能接收一行文本,因此PRINT程序发送一行文本后,必须等待该行被打印出来以后才能发送另一行,如此重复直至所有结果都被打印出来。这种简单方法的缺点是处理器要花费相当多的时间等待打印机准备就绪。如果可以将打印和计算过程重叠,在打印过程中执行COMPUTE程序,那么总的执行速度就会快得多。可以按照下面的方法实现。首先,执行COMPUTE程序产生第一个 $n$ 行的输出;然后执行PRINT程序发送第一行文本给打印机。此时,我们不等待该行被打印出来,将PRINT程序暂时中断转去执行COMPUTE程序;当打印机就绪后,发送一个“中断请求”信号通知处理器。处理器响应后,中断COMPUTE程序并将控制权转给PRINT程序;PRINT程序将第二行发送给打印机并再次中断,接着被中断的COMPUTE程序在中断点处重新开始执行。这个过程将持续下去直到所有 $n$ 行都被打印出来,PRINT程序结束。

在下一个 $n$ 行输出准备就绪时,PRINT程序将重新启动。如果COMPUTE产生 $n$ 行输出花费的时间要长于打印的时间,那么处理器将一直执行有效的计算。

这个例子阐明了中断的概念。响应中断请求时执行的程序称为中断服务程序,这里PRINT程序便是中断服务程序。中断与子程序调用十分相似。假设在执行图4-5中的指令 $i$ 时有一个中断请求到达。那么处理器先执行完指令 $i$ ,然后将中断服务程序的第一条指令地址装入程序计数器。这里,我们不妨假设该地址就在处理器中。执行完中断服务程序后,处理器返回到指令 $i+1$ 处。因此,当发生中断请求时,当前PC的内容指向第 $i+1$ 条指令,并被暂时保存到一个已知区域中。在中断服务程序结束后,中断返回指令从暂时保存区域取出保存的内容并重新装入到PC中,处理器开始在第 $i+1$ 条指令处接着执行。许多处理器的返回地址保存在堆栈中。此外,返回地址还可以保存在一个专门区域,比如专门提供的寄存器中。

[208]

应该注意的是,作为中断处理的一部分,处理器必须通知设备它的请求已经被识别,使得该设备撤销它的中断请求信号。这可以由总线上的专门控制信号来实现。在后面将要讲述的中断方式中使用的中断确认信号就是用来实现这一功能的。此外,通过处理器与I/O设备接口之间

传输数据也是一种通用的方法。在中断程序中，当执行访问设备接口状态和数据寄存器的指令时，也就相当于通知设备中断请求已经被识别了。

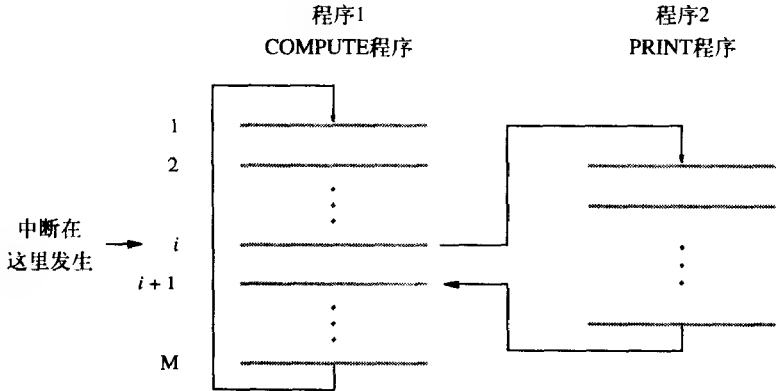


图4-5 使用中断方式转移控制

对中断程序的处理和子程序十分相似。但需要注意的是二者有着重要的区别。子程序被调用时执行的是调用程序请求的功能。而中断服务程序与收到中断请求时正在执行的程序毫不相[209]关，实际上两个程序常常属于不同的用户。因此在执行中断服务程序之前，任何在程序执行期间可能被改变的信息都必须保存起来。这些信息在被中断程序重新开始执行时必须重新装入。这样，初始的程序就可以继续执行，除了时间延迟以外不受中断的任何影响。需要保存和恢复的信息一般包括：状态码标志、所有中断服务程序和被中断程序都使用的寄存器内容。

保存和恢复信息的任务可以由处理器自动完成，也可以由程序指令完成。大部分现代处理器只保存能保持程序完整执行的最小信息量。这是因为保存和恢复寄存器的过程中包括存储器传递，它将增加总的执行时间，增加执行开销。并且，保存寄存器也会增加从接收中断请求到开始执行中断服务程序之间的时间延迟。这一延迟称为中断等待。在一些应用中，过长的中断等待是不可接受的。所以，应尽量减少处理器自动保存的信息量。通常，处理器只保存程序计数器和处理器状态寄存器的内容。任何需要保存的额外信息都必须由程序指令在中断处理程序开始时保存起来，在中断处理完成后进行恢复。

在一些早期的处理器中，尤其是在那些只有很少寄存器的处理器中，收到中断请求时所有寄存器都被处理器自动保存起来。处理完成后，由中断返回指令将这些保存的信息恢复到各自的寄存器中。有一些计算机提供了两种类型的中断，其中一种需要保存所有寄存器的内容，另一种不需要。I/O设备可以随意选择，这取决于对响应时间的要求。还有一种方式是复制处理器的寄存器组。在这种方式中，中断服务程序可以使用另一个不同的寄存器组，不需要保存和恢复寄存器。

中断要比简单的程序控制能更好地协调I/O传输。从一般意义上说，中断能够由外部事件引起控制权从一个程序转移到另一个程序。被中断的程序在中断服务程序执行完后接着继续执行。在操作系统和很多其他控制应用中都使用了中断的概念。在这些应用中，特定程序的运行必须与外部事件保持精确的时间关系，这通常称为实时处理。

4.2.1 中断的硬件

I/O设备通过激活总线上的中断请求线来请求中断。大部分计算机中都有多台可以请求中断

的设备，因此单根中断请求线可能要被 $n$ 台设备共用，如图4-6所示。所有设备都通过接地开关连接到中断请求线上。当要请求一次中断时，设备闭合与之相连的开关。如果所有中断请求信号 $INTR_1$ 到 $INTR_n$ 都是无效的，即所有开关都是断开的，中断请求线上的电平就等于 $V_{dd}$ ，此时中断请求线处于无效状态。当一台设备闭合开关请求中断时，中断请求线上的电平降为0，处理器接收到的中断请求信号 $\overline{INTR}$ 就变成了1。由于闭合一个或更多开关都将导致请求线上的电平降为0，所以 $\overline{INTR}$ 的值是所有设备的中断请求的逻辑“或”，即：

$$\overline{INTR} = INTR_1 + \cdots + INTR_n$$

习惯上常用求反的形式 $\overline{INTR}$ 来命名公共线上的中断请求信号，因为这个信号是低电平有效的。

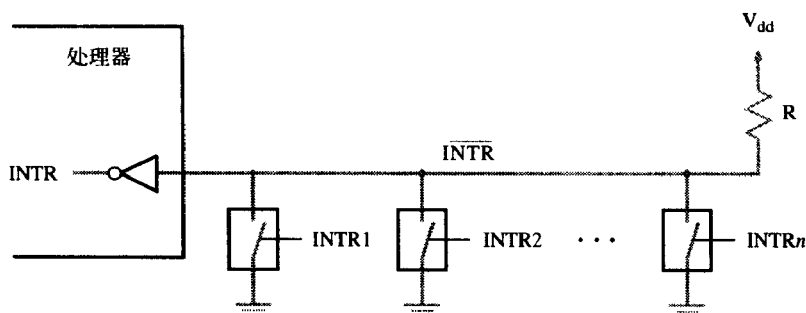


图4-6 公共中断请求线的漏极开路总线等效电路图

图4-6的电路中，使用专用的集电极开路（open-collector）门（双极型电路中）或漏极开路（open-drain）门（MOS型电路中）来驱动 $\overline{INTR}$ 线。集电极开路门或漏极开路门的输出与接地开关相同，门电路的输入为0时开关断开，门电路的输入为1时开关闭合。门电路的输出电压（即逻辑状态）由所有与总线相连的门电路上的数据按照上面给出的等式计算得到。电阻 $R$ 称为上拉电阻，因为当开关断开时它将线路拉升到高电平状态。

#### 4.2.2 中断的允许和禁止

计算机所提供的设施必须使程序员能够完全控制程序执行期间发生的事件。当外部设备中断请求到达时，处理器将中止一个程序的执行转而去启动另一个程序。由于随时都可能出现，所以中断有可能改变程序员预先设定的执行顺序。因此，必须谨慎处理程序执行中的中断。计算机的一个基本功能就是能按照需要允许和禁止这些中断。下面将详细分析一下该功能及其相关的设备。

在很多情况下处理器需要忽略中断请求。例如，图4-5的Compute-Print程序中，打印机的中断请求只有存在需要打印的输出行时才能被接受。在打印完一组 $n$ 行中的最后一行后，中断就应该被禁止，直到另一组准备就绪。在有些情况下，可能需要保证特定的指令序列没有中断地执行下去，因为中断服务程序可能会改变序列中某些指令所需要的数据。必须提供一些允许和禁止中断的方法供程序员使用。其中一种简单的方法就是提供执行这些功能的机器指令，如允许中断和禁止中断指令。

下面具体考虑只有一台设备的单一中断请求的特殊情况。当一台设备激活中断请求信号时，它保持这个信号直到确认处理器已经接收了该中断请求。这就是说中断请求信号在中断服务程序执行期间将保持有效，也许要持续到访问该设备的指令到达。另一方面也要保证这个有效的

请求信号不会导致连续的中断，使系统进入死循环。解决这个问题的方法有几种。这里将描述其中的三种可行方法，可同时处理多个中断设备的方法将在后面讲述。

第一种可行方法是使处理器硬件忽略中断请求，直至中断服务程序的第一条指令执行完毕。然后，使用一条禁止中断指令作为中断处理程序的第一条指令，这样程序员就可以保证在执行允许中断指令前不会发生更多的中断。允许中断指令一般是中断服务程序中中断返回指令之前的最后一条指令。处理器必须保证在下一次中断出现之前执行完中断返回指令。

第二种方法是使处理器在开始执行中断服务程序前自动禁止中断，这种方法适用于只有一根中断请求线的简单处理器。在将PC和处理器状态寄存器（PS）的内容保存堆栈后，处理器执行与禁止中断指令等效的操作。在实际中，我们常常用PS寄存器中的一位，称为中断允许位，来指示中断是否被允许。在该位为1时接收到的中断请求将被接受。将PS的内容保存堆栈后，此时的中断允许位为1，然后处理器将PS寄存器中的中断允许位清0，从而禁止新的中断发生。当执行中断返回指令时，PS寄存器从堆栈中恢复中断前的内容，中断允许位被重置1。这样，中断被重新允许。

第三种方法，在处理器中有一根特殊的中断请求线，它的中断处理电路只在信号的前沿响应，称为边沿触发（edge-triggered）。在这种情况下，不论这根线的有效状态持续多久，处理器都只接收到一个中断请求。因此就没有多次中断的危险，也不需要明确地将这根线禁止中断。

在继续学习更复杂的中断内容前，总结一下处理来自单一设备的中断请求时包含的事件序列。假设是允许中断的，下面是一个一般的过程：

1. 设备发出一个中断请求。
2. 处理器中止当前执行的程序。
3. 通过改变PS寄存器中控制位的值禁止中断（边缘触发中断除外）。
4. 通知设备请求已被识别，设备将中断请求信号撤销。
5. 中断服务程序执行中断请求的操作。
6. 允许中断，被中断程序继续执行。

212

#### 4.2.3 处理多台设备

在处理器连接多台能够发生中断请求的设备时，因为在操作上这些设备是相互独立的，所以没有固定的中断发生顺序。例如，正在处理设备Y的中断时设备X可能又发出中断请求，或者几台设备都在同一时刻发出中断请求。这导致了一系列的问题：

1. 处理器如何识别发出中断请求的设备？
2. 如果不同的设备要求不同的中断服务程序，在每一种情况下处理器怎样获取适当的程序起始地址？

3. 当有中断正在被处理时，另一台设备是否可以中断处理器？
4. 怎样处理两个或更多同时产生的中断请求？

不同计算机处理这些问题的方法各不相同，解决方法是决定计算机和给定应用相适应与否的重要因素。

在图4-6中，公共中断请求线上收到一个请求后，需要附加信息来识别是哪一台设备发出的请求。而且，如果有两台设备同时发出请求，必须可以选择其中一个进行处理。执行完选中设备的中断服务程序后，处理器可以响应第二个中断请求。



可以通过读取状态寄存器中的信息来确定一个设备是否正在请求中断。当一台设备发出中断请求后，它的状态寄存器中的一位被置成1，这一位称为IRQ位。图4-3中的KIRQ和DIRQ位分别是键盘和显示器的中断请求位。最简单的识别中断设备的方法就是用中断处理程序测试所有与总线相连的I/O设备。测试到的IRQ位被置1的第一个设备就是发出中断请求的设备，相应的处理子程序将被调用。

213 轮询方式非常容易实现。它的主要缺点是：设备即使没有发出任何服务请求，系统也要花费时间去检查IRQ位。另一种可供选择的方法是使用向量中断。

### 向量中断

为了节约检测过程所花费的时间，请求中断的设备可以直接向处理器标明它自己。然后，处理器就可以立即开始执行相应的中断服务程序。术语向量中断指的是所有基于这种方法的中断处理方式。

请求中断的设备可以通过总线向处理器发送一个专门的代码来标识自己。这样，即使使用同一根中断请求线，处理器仍可以识别出每台设备。设备提供的代码可以表示它的中断服务程序的起始地址。代码长度一般为4到8位，起始地址的其余部分由处理器根据它在存储器中存放中断服务程序地址的区域提供。

这种工作方式要求给定设备的中断服务程序必须总是在同一位置开始。不过程序员可以在该位置放一条指令来提高灵活性，由该指令负责跳转到相应的中断服务程序。在许多计算机中，这是由中断处理机制自动完成的。中断设备所指向的位置用来存储中断服务程序的起始地址。处理器读取该地址，称为中断向量，并装入到PC寄存器中。中断向量也可以包含处理器状态寄存器的新值。

大部分计算机的I/O设备在数据总线上发送中断向量代码，并使用总线控制信号来确保设备之间互不干扰。当一台设备发送中断请求时，处理器可能不能立即接收中断向量代码。例如，处理器必须先执行完当前的指令，而这条指令可能要求使用总线。而且，如果发出请求时恰巧该中断被禁止，就可能会有更长的延迟。中断设备必须等到处理器准备接收时才能将数据放到总线上。当处理器准备好后，就激活中断确认线INTA。然后I/O设备发送自己的中断向量代码进行响应，并关闭INTR信号。

### 中断嵌套

214 在4.2.1节中我们提到在执行中断服务程序期间应该禁止中断，以保证同一设备的中断请求不会导致多次中断。这一方式还常用于多台设备的情况，在这种情况下特定中断服务程序一旦开始执行，处理器就必须等到服务程序执行完毕才能接受第二个设备的中断请求。但中断服务程序一般都很短，导致的延迟对于大部分设备来说都是可以接受的。

然而，在某些设备中，响应中断请求时过长的延迟将会导致错误的操作。例如，有一台使用实时时钟记录日常时间的计算机。实时时钟在固定的时间间隔内向处理器发送中断请求。对每一个请求，处理器执行一个简短的中断服务程序来增加存储器中的计数器组，这些计数器用秒、分等来记录时间。正确的操作要求响应实时时钟中断请求的延迟必须小于两个连续请求之间的时间间隔。为了确保在存在其他中断设备的情况下满足这一条件，需要在执行其他设备中断服务程序时接收时钟的中断请求。

这个例子表明I/O设备需要按照一定优先级结构组织起来。在处理低优先级设备的中断时，高优先级设备的中断请求应该被接受。

多优先级结构意味着在执行中断服务程序期间,一些设备的中断请求可以接受,而其他一些设备的请求不可以接受,这取决于设备的优先级。为了实现这种方式,我们可以给处理器分配一个优先级,它可以由程序控制改变。处理器的优先级就是当前执行的程序的优先级,处理器只接受优先级高于它的设备的中断。在设备的中断服务程序开始执行时,处理器的优先级提升为该设备的优先级。这一操作将对同级或较低级的设备禁止中断,而来自更高优先级设备的中断请求将继续被接受。

处理器的优先级通常编码在处理器状态字的某一些位中,可以被写入PS寄存器的程序指令改变。这些指令都是特权指令,只有当处理器在管态模式下运行时才能执行。处理器只有在执行操作系统程序时才处于管态模式。它在开始执行用户程序前转换为用户模式。因此,用户程序无法有意或无意地改变处理器的优先级而使操作系统崩溃。在用户模式下试图执行特权指令将会导致一种特殊类型的中断,称为特权异常,我们将在4.2.5节讲述这些内容。

多优先级模式可以通过对每台设备使用单独的中断请求和中断确认线来实现,如图4-7所示。每根中断请求线分配不同的优先级。在这些线上收到的中断请求将发送到处理器的优先级仲裁电路上。只有当优先级高于处理器的优先级时,请求才被接受。

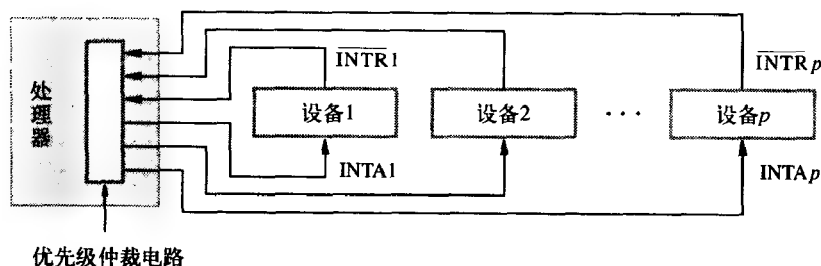


图4-7 使用独立的中断请求和确认线实现中断优先级

215

### 同时请求

现在考虑两台或多台设备的中断请求同时到达的问题。处理器必须决定哪个请求最先被服务。如果采用图4-7所示的方式,那么很容易解决。只需简单地接受优先级最高的请求即可。然而如果几台设备共享一条中断请求线,如图4-6所示,则需要其他机制来解决该问题。

轮询I/O设备的状态寄存器是最简单的方法。这种方式下,优先级由轮询设备的顺序来决定。当使用向量中断时,必须保证只有一台设备被选定发送它的中断向量代码。一种广泛使用的方法是将设备连接起来形成一个菊花链(daisy chain),如图4-8a所示。中断请求线 $\overline{INTR}$ 被所有设备共用。中断确认线INTA以菊花链的形式连接起来,保证INTA信号以串行方式传递到所有设备上。当有几台设备提出中断请求时, $\overline{INTR}$ 线有效,处理器响应将INTA线置为1。INTA信号将被设备1接收到,如果设备1没有请求任何服务,它就将信号传递给设备2。如果设备1有未被响应的中断请求,便将阻断INTA信号并接着将它的识别码放到数据线上。因此,在菊花链结构中,电路上最接近处理器的设备的优先级最高。沿该链次接近的设备有第二高的优先级,其他的依次类推。

图4-8a所示方式需要的线路数量比图4-7中的单独连接少得多。图4-7所示方式的主要优点是,它可以使处理器根据设备的优先级有选择地接受中断请求。将这两种方式结合起来我们可以得到图4-8b中更加通用的结构。设备以组为单位组织起来,每个组有不同的优先级。在同一个组内的设备以菊花链形式连接起来。很多计算机系统使用的就是这种结构。

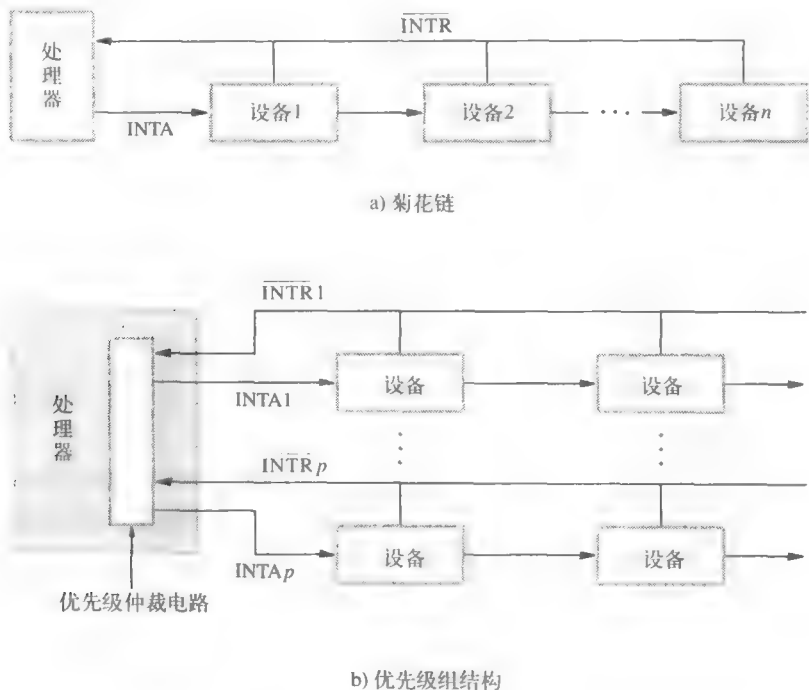


图4-8 中断优先级方式

216

#### 4.2.4 控制设备请求

前面我们假设当I/O设备接口准备一次传送时，它只产生一个中断请求。例如在图4-3中的SIN标志等于1时。确保中断请求只来自那些正被程序使用的设备是非常重要的。闲置设备不允许产生中断请求，即使它们已经准备好参与I/O传送操作。因此，我们需要一个机制，在每台设备的接口电路控制该设备是否允许产生中断。

这一控制在设备接口电路中通常以中断允许位的形式提供。在图4-3的CONTROL寄存器中，键盘的中断允许位KEN标志和显示器的中断允许位DEN标志就是用来完成该功能的。如果其中一个标志被置位，当STATUS寄存器中相应的状态标志也被置位时，接口电路就产生一个中断请求。同时，当接口电路将KIRQ或DIRQ分别置位表示键盘或显示器设备正在请求中断时，如果中断允许位为0，接口电路将忽略状态标志的状态，不产生中断请求。

总的来说，有两个独立的机制来控制中断请求。在设备端，控制寄存器中的中断允许位决定设备是否被允许产生中断请求。在处理器端，PS寄存器中的中断允许位或优先级结构决定特定的中断请求是否被响应。

**例4.3** 一个使用中断向量方式的处理器，中断服务程序的起始地址存储在存储器的INTVEC单元处。将处理器状态字中的中断允许位IE（假定它是第9位）置1，则允许中断。与处理器相连的键盘和显示器单元具有图4-3所示的状态、控制和数据寄存器。

假设我们希望在主程序的某处从键盘输入一行字符，并存储到存储器中从LINE单元开始的连续字节区域。如果使用中断来执行这项操作，需要初始化中断过程。这一过程可以按照如下步骤来实现：

217

1. 将中断服务程序的起始地址装入INTVEC单元。  
 2. 将地址LINE装入存储器中的PNTR单元。中断服务程序将使用这个单元作为指针并将输入字符存储到存储器中。

3. 通过将CONTROL寄存器中第2位置1来允许键盘中断。

4. 将处理器状态寄存器PS中的IE位置1, 允许中断。

该初始化过程完成后, 从键盘输入一个字符就会导致键盘接口产生一个中断请求。此时正在执行的程序将被中断, 转去执行键盘输入中断服务程序。该中断服务程序必须执行以下任务:

1. 从键盘的输入数据寄存器中读取输入字符。然后, 接口电路撤销中断请求。

2. 将字符存储到PNTR指向的存储器位置, 并递增PNTR。

3. 当到达行尾时, 禁止键盘中断并通知主程序。

4. 从中断返回。

用来执行这些任务的指令如图4-9所示。当检测到输入行的末端时, 中断服务程序将CONTROL寄存器中的KEN位清除, 表示不期望更多的输入了。同时也将变量EOL (End Of Line) 置1, 它在初始时被置0。假定主程序用周期性的查询来确定输入行是否准备好进行处理。

<b>主程序</b>			
	Move	#LINE,PNTR	初始化缓冲区指针
	Clear	EOL	清除行末指示器
	BitSet	#2,CONTROL	允许键盘中断
	BitSet	#9,PS	设置PS中的中断允许位
	⋮		
<b>中断服务程序</b>			
READ	MoveMultiple	R0-R1,-(SP)	保存寄存器R0和R1到堆栈中
	Move	PNTR,R0	载入地址指针
	MoveByte	DATAIN,R1	获取输入字符并保存到存储器
	MoveByte	R1,(R0)+	
	Move	R0,PNTR	更新指针
	CompareByte	#\$0D,R1	检查是否为回车符
	Branch ≠ 0	RTRN	
	Move	#1,EOL	指示行末
	BitClear	#2,CONTROL	禁止键盘中断
RTRN	MoveMultiple	(SP)+,R0-R1	恢复寄存器R0和R1
	Return-from-interrupt		

图4-9 利用图4-3中的寄存器使用中断从键盘读入一行字符

计算机中的输入/输出操作通常要比这个例子复杂得多。正如我们将在4.2.5节中描述的, 计算机操作系统在用户级程序上执行这些操作。

#### 4.2.5 异常

中断是导致一个程序的执行被中止、另一个程序的执行开始的事件。到目前为止, 我们处理的中断只是由那些在I/O数据传输期间接收到的请求而引起的。然而, 中断机制还可以用在许多其他的情形下。

术语异常通常指任何引起中断的事件。因此, I/O中断是异常的一种。现在我们来讲述几种

其他类型的异常。

### 错误恢复

计算机使用大量的技术来确保所有硬件都能正确工作。例如,许多计算机在主存中有错误校验码,它可以监测存储的数据是否正确。如果出现了错误,控制硬件就会检测到并产生中断通知处理器。

如果检测到错误或不正常情况,即便此时处理器正在执行某个程序的指令,也会中断该程序。

[218] 例如,一条指令的OP码字段有可能不对应任何合法指令,或一条算术指令试图除以0。

当这些错误引发异常时,处理器按照与处理I/O中断请求相同的方式进行处理,中止正在执行的程序并启动异常服务程序。这个程序进行适当的操作,尽可能从错误中恢复过来,或将该错误通知给用户。回想一下I/O中断时的情形,我们知道处理器在接受中断前必须完成当前指令。然而,当中断是由错误引起的时候,被中断的指令通常不等执行完,处理器就转去处理异常。

### 调试

另一种非常重要的异常用于帮助调试程序。系统软件通常包括一个调试器程序,它用来帮助程序员找出程序中的错误。调试器使用中断来提供两项重要的功能:跟踪和断点。

当处理器运行在跟踪模式下时,每执行完一条指令就产生一次异常,异常服务程序便是调试程序。调试程序使用户可以查看寄存器、存储器等内容。从调试程序返回后接着执行被调试程序的下一条指令,然后再次转到调试程序。但在执行调试程序期间是禁止跟踪异常的。

[219]

断点除了被调试程序只在用户选择的特定点发生中断外,也提供了与跟踪相似的功能。陷阱和软件中断指令用来实现断点调试。执行这些指令和接收硬件中断请求是等效的。当调试一个程序时,用户可能希望在指令 $i$ 后中断程序的执行。调试程序保存指令 $i+1$ ,并用一条软件中断指令取代它。当程序执行到该点时就发生中断,转入调试程序。这就给了用户一次检查存储器和寄存器内容的机会。当用户准备好继续执行被调试程序时,调试程序恢复被保存的指令 $i+1$ ,并执行一条中断返回指令。

### 特权异常

为了保护计算机操作系统不被用户程序破坏,一些特定的指令只有当处理器处于管态模式时才允许执行。这些指令称为特权指令。例如,当处理器在用户模式运行时,改变处理器优先级的指令或用户访问计算机存储器中其他区域的指令将不能执行。试图执行这样的指令将产生特权异常,导致处理器转入到管态模式并开始执行操作系统中的相应程序。

## 4.2.6 在操作系统中使用的中断

操作系统(OS)的职责是协调计算机内部所有的活动。它在执行I/O操作、与用户程序通信和控制用户程序时使用了大量的中断。中断机制使操作系统可以分配优先级、从一个用户程序转换到另一个用户程序、实现安全和保护特性、协调I/O活动。后面章节将对这些做简要讨论,而本书不是专门介绍操作系统的,我们的目标是说明如何使用中断。

操作系统将所有与计算机相连的设备的中断服务程序组织在一起。应用程序本身不执行I/O操作。当应用程序需要进行I/O操作时,它首先指向需要传输的数据,然后请求OS执行操作。OS暂停执行该程序转去执行请求的I/O操作。当操作结束后,OS将控制权传回给应用程序。OS和应用程序之间的控制权传递是使用软件中断来实现的。

[220]

操作系统为应用程序提供各种各样的服务。为了便于实现这些服务,大部分操作系统都有几条不同的软件中断指令,每条指令都有自己的中断向量。根据被请求的服务,它们可以调用OS

的不同部分。但是,有的处理器可能只有一条软件中断指令,该指令使用一个立即操作数来表示特定的服务。

在具有管理员和用户两种模式的计算机中,接到一个中断请求时,处理器便切换到管态模式下。处理器首先将寄存器的内容保存到堆栈里,然后设置处理器状态寄存器中的某一位来实现这一切换。因此,当一个应用程序通过软件中断指令调用OS时,处理器将自动切换到管态模式,这时OS可以访问所有的计算机资源。OS执行中断返回指令后,应用程序的处理器状态字从堆栈中恢复过来,处理器切换回用户模式。

为了说明应用程序和操作系统之间的相互作用,我们来看一个多任务的例子。多任务是处理器同时执行多个用户程序的操作模式。其中,通用的实现这一模式的一种OS技术是时间片(time slicing)。采用这种技术,每个程序只运行一段很短的时间,称为时间片 $\tau$ ,然后另一个程序接着运行它的时间片,如此继续下去。周期 $\tau$ 由连续运行的硬时钏决定,这个时钟每 $\tau$ 秒钟产生一个中断。

图4-10描述了在多任务环境下实现一些必要功能所需的程序。操作系统启动时要执行一个初始化程序,在图中称为OSINIT。在这个过程中,初始化程序将适当的值装入到存储器的中断向量区域。这些值对应于不同中断的中断服务程序起始地址。例如,OSINIT将SCHEDULER程序的起始地址装入到定时器中断的中断向量中。因此,在每一个时间片结束时,定时器中断将启动该程序执行。

程序与所有描述它当前执行状态的信息一起被操作系统视为一个整体,称为进程。一个进程有三种状态:运行、就绪和阻塞。运行状态指的是程序当前正在被执行。如果程序已经准备好执行但仍在等待被调度程序选定,那么此时进程就处于就绪状态。第三种状态阻塞指的是程序因为一些原因而没有准备好开始执行。比如,它可能正在等待所请求的I/O操作的完成。

假设程序A在一个特定时间片内处于运行状态。在该时间片结束时,定时器中断这个程序的执行并开始执行SCHEDULER。SCHEDULER是一个操作系统例行程序,它来决定哪个用户程序占用下个时间片。该程序首先保存以后继续执行程序A所需要的全部信息。这些保存的信息称为程序状态,包括寄存器的内容、程序计数器和处理器状态字。保存寄存器的内容是因为它可能包括中断发生时一些正在进行的计算的中间结果。程序计数器指向下一次开始执行的位置。处理器状态字也需要保存,因为它包含着状态码标志以及其他信息,比如优先级等。

然后,SCHEDULER选择执行另一程序B,程序B在早些时候被中断并正处于就绪状态。SCHEDULER将B程序中断时保存的所有信息恢复,包括PS和PC的内容,然后执行一条中断返回指令。于是,程序B就重新开始执行 $\tau$ 秒钟,在这个时间片结束时定时器时钟再次产生一个中断,并产生到另一个就绪进程的上下文切换。

假设程序A需要从键盘读入一个输入行。它并不亲自执行这个操作,而是向操作系统请求I/O服务。它使用堆栈或处理器寄存器向OS传递信息来描述请求的操作、I/O设备和程序数据区域中存放输入行的缓冲区地址。然后执行一条软件中断指令,这条指令的中断向量指向图4-10a中的OSSERVICES程序。OSSERVICES程序检查堆栈中的信息并调用适当的OS程序启动请求的操作。在此例中,它将调用图4-10b中的IOINIT程序,该程序负责启动I/O操作。

当进行I/O操作时,请求操作的程序不能继续执行。因此,IOINIT程序将与程序A相关联的进程设置成阻塞状态,向调度程序表明这个程序此时不能继续执行。IOINIT程序执行I/O操作需要的所有准备工作,如初始化地址指针和字节计数器,然后调用一个程序执行I/O的传输。

OSINIT	设置中断向量 时间片时钟 软件中断 键盘中断	← SCHEDULER ← OSSERVICES ← IODATA
OSSERVICES	检查栈, 确定请求的操作 调用适当的程序	
SCHEDULER	保存程序状态 选择一个就绪进程 恢复保存的新进程的上下文 将PS和PC的新值压入栈中 返回中断	

a) OS初始化、服务程序和调度程序

IOINIT	进程状态设置为“阻塞”状态 初始化存储器缓冲区地址指针和计数器 调用设备驱动程序, 初始化设备并在设备接口允许中断 从子程序返回
IODATA	轮询设备, 确定中断源 调用适当的驱动程序 如果END=1, 则设置进程状态为“就绪” 中断返回

b) I/O程序

KBDINIT	允许中断 从子程序返回
KBDDATA	检查设备状态 如果就绪, 便传输字符 如果字符=CR, 则(设置END=1; 禁止中断), 否则设置END=0 从子程序返回

c) 键盘驱动程序

222

图4-10 几个操作系统程序

设计操作系统时普遍使用的方法是将所有属于特定设备的软件封装成一个独立的模块, 称为设备驱动程序。这样的模块可以很容易地添加到OS中或从OS中删除。在该例子中键盘的设备驱动程序由两个程序组成: KBDINIT和KBDDATA, 如图4-10c所示。IOINIT程序调用KBDINIT, KBDINIT用来初始化设备和接口电路所需要的信息, 同时KBDINIT还通过在接口电路中设置控制寄存器的相应位来允许中断, 然后返回IOINIT, IOINIT再返回到OSSERVICES。至此键盘已准备好数据传输操作了。它将在一个键被按下时产生中断请求。

返回到OSSERVICES后, SCHEDULER程序选择另一个用户程序运行。当然, 调度程序不会选择程序A, 因为它正被阻塞。中断返回指令不仅能使被选定的用户程序开始执行, 同时还能通过向处理器状态寄存器中装入新值来允许中断。这样, 键盘接口电路产生的中断请求将被接受, 该中断的中断向量指向一个称为IODATA的OS程序。因为可能会有几台设备连接到同一根中断请

求线上，所以IOWDATA首先测试这些设备，以确定发出请求的设备，然后调用适当的设备驱动程序来为该请求服务。在我们的例子中，调用的驱动程序是KBDDATA，它将传输一个字符的数据。如果是回车符，它还将END标志置为1来通知IOWDATA此次I/O操作已经完成。同时，IOWDATA程序将进程A的状态由阻塞变为就绪，调度程序在未来的时间片中便可以选

223

4.3 处理器举例

前面我们概括地讲述了中断机制。商业处理器提供了很多前面描述的特性和控制机制，但并不是所有的机制都是必需的。例如向量中断使得处理器能够迅速跳转到特定设备的中断服务程序。但除此之外，识别设备和确定适当的中断处理程序起始地址的任务也可以由软件使用轮询方法来实现。下面我们将讲述第3章中所描述的三种处理器的中断处理机制。

4.3.1 ARM中断结构

ARM处理器有一个简单但强大的异常处理机制。有5种类型的异常，其中只有两种是外部中断请求线：IRQ和FIQ（快速中断请求）。还有一条软件中断指令SWI和两种由程序执行期间遇到的不正常情况引起的异常。这两种异常分别是总线错误引起的外部故障和试图执行未定义的指令。异常按照下面的优先级进行处理：

- 1. 复位（最高优先级）
- 2. 数据中止
- 3. FIQ
- 4. IRQ
- 5. 预取中止
- 6. 未定义指令（最低优先级）

复位条件被包括在这个结构中，是因为它必须超越所有其他条件将处理器恢复到一个已知的初始条件下。还注意到有两种中止条件：数据中止（Data Abort）是由读写错误引起的，预取中止（Prefetch Abort）是由从存储器中预取指令时发生的错误引起的。

图3-1显示了ARM处理器的状态寄存器CPSR（Current Program Status Register，当前程序状态寄存器）。该寄存器低位字节如图4-11所示。其中有两个中断屏蔽位，分别对应IRQ和FIQ。当其中一位等于1时，相应的中断就被禁止了。这个寄存器还包括5个模式位M4-0，指示处理器正在运行的模式。ARM共有6个模式——1个用户模式和5个特权模式（分别对应5种异常）。

当处理器转换到不同模式时，同时也要转换程序可访问的某些寄存器。每种模式下能访问的寄存器组如图4-12所示。寄存器R0到R7、R15（PC）和CPSR在所有模式下都可以访问。在特权模式下，除FIQ、R8到R12也都是可以访问的。在IRQ、管态、中止和未定义模式下均有两个新的寄存器取代R13和R14。在FIQ模式中，寄存器R8到R14被R8\_fiq到R14\_fiq代替。取代用户模式寄存器的寄存器称为保留寄存器。它们可以被中断服务程序使用，因而就不需要保存用户模式下对应寄存器的内容。例如，在IRQ模式下当一条指令访问寄存器R13时，被访问的寄存器是R13\_irq而不是用户模式下的寄存器R13。而且，除用户模式外的每个模式都有一个专用的寄存器，称为保存处理器状态寄存器（如SPSR\_svc，SPSR\_irq等）。当发生中断时，它用来保存寄存器CPSR的内容。

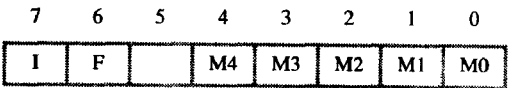


图4-11 ARM处理器状态寄存器的低位字节

224



通用寄存器和程序计数器

用户	FIQ	IRQ	管态	中止	未定义
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
R15	R15	R15	R15	R15	R15

处理器状态寄存器

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

225

图4-12 ARM处理器不同模式下可以访问的寄存器

异常处理程序总是从存储器中的固定位置启动，如表4-1所示。中断之后，处理器进入指定的模式并在相应的向量地址开始执行程序。由于除了最后地址（FIQ）以外只有一条指令空间，所以该位置应该包括跳转到服务程序的指令。在FIQ模式下，服务程序不需要使用跳转指令就可以从所示的起始位置继续执行。

表4-1 ARM处理器中的中断向量地址

地 址 (16进制)	异 常	进入的模式
0	复位	管态
4	未定义指令	未定义
8	软件中断	管态
C	预取中止	中止
10	数据中止	中止
14	保留	
18	IRQ	IRQ
1C	FIQ	FIQ

当处理器收到一个中断时，它执行以下操作：

1. 保存被中断程序的返回地址到相应模式下的寄存器R14中。例如，在FIQ模式下，它将返回地址保存到寄存器R14\_fiq中。保存的具体值依赖于中断类型，我们将会进行简要的解释。
2. 保存处理器状态寄存器CPSR的内容到相应的SPSR中。
3. 根据中断类型改变CPSR中的模式位，如表4-1中最后一列所示。对于FIQ和IRQ两种模式，处理器还将CPSR中相应的屏蔽位置1以禁止同一条线上更多的中断。
4. 跳转到从相应向量地址开始的中断服务程序。

226

ARM处理器采用流水线结构，我们将在第8章中详细解释。这种结构指的是在前一条指令执行完之前就开始预取下一条指令。假设处理器在某个地址A处取得指令 $I_1$ 。处理器将PC寄存器中的值增加到A + 4并开始执行指令 $I_1$ 。在执行完这条指令前，它从地址A + 4处预取得指令 $I_2$ ，然后将PC的值增加到A + 8。现在假设在指令 $I_1$ 执行结束时处理器检测到一个IRQ中断，于是处理器就开始执行上面描述的操作。它将PC中的内容复制到寄存器R14\_irq中，此时PC的值为A + 8。这样已经取出但没有执行的指令 $I_2$ 就被放弃了，但这个指令是中断服务程序必须返回到的指令。

在上面讲述的情况中，保存到R14\_irq中的值是A+8，但中断服务程序返回的地址必须是A + 4。这就是说中断程序在R14\_irq中的内容作为返回地址之前必须将其减去4。也就是说，返回指令必须将值[R14\_irq] - 4装入到PC中。当然，也必须将SPSR\_irq的内容复制到CPSR中。后一操作将处理器恢复到中断发生前的运行模式，并且也清除了中断屏蔽并再次允许中断。要求的操作可由指令

SUBS PC, R14\_irq, #4

来执行。这条指令从R14\_irq中减去4然后将结果存储到R15中。后缀S通常指的是设置状态码。当这条指令的目标寄存器是PC时，后缀S使处理器将SPSR\_irq中的内容复制到CPSR中，从而完成回到被中断程序所需要的操作。

为了获得正确的返回地址，从R14中减去的数值大小依赖于处理器流水线指令执行的具体情况。因此，对于不同的异常这个数值是不同的。例如，在由SWI指令触发的软件中断中，保存在R14\_svc中的值本身就是正确的返回地址。因此，从SWI服务程序返回可以由指令

MOVS PC, R14\_svc

完成。表4-2给出了表4-1中每种异常的正确返回地址和回到被中断程序的指令。需要注意的是，对于中止中断，可能是由于总线错误导致的，表中显示的期望返回地址是导致该错误指令的地址。这里假设控制软件希望重新执行该指令。

在特权模式下运行时，有两条专门的MOV指令MSR和MRS传送来自或发送给PSR的数据，这些数据可以是PSR的当前数据或者保存在PSR中的数据。例如

MRS R0, CPSR

将CPSR的内容复制到R0。相似地，

MSR SPSR, R0

将寄存器R0中的内容复制到SPSR中。这些指令在操作系统需要允许/禁止中断时非常有用，在后面例4.4中我们将看到这一点。

227

表4-2 异常返回时的地址修正

异 常	保存的地址 <sup>①</sup>	正确的返回地址	返回指令
未定义指令	PC+4	PC+4	MOVS PC, R14_und
软件中断	PC+4	PC+4	MOVS PC, R14_svc
预取中止	PC+4	PC	SUBS PC, R14_abt, #4
数据中止	PC+8	PC	SUBS PC, R14_abt, #8
IRQ	PC+4	PC	SUBS PC, R14_irq, #4
FIQ	PC+4	PC	SUBS PC, R14_fiq, #4

① PC是导致异常的指令的地址。在IRQ和FIQ中，PC是由于中断而没有被执行的第一条指令的地址。

### 堆栈和嵌套

ARM的中断机制将返回地址存储在一个寄存器中，而不是自动实现堆栈机制来允许子程序或中断嵌套。这样做对程序员有如下好处：当需要时可以实现该功能，当不需要时可避免不必要的开销。首先，不同中断源可以引起中断嵌套。例如，IRQ中断服务程序的返回地址在R14\_irq中，可能被优先级更高的FIQ中断。新的返回地址将被保存在R14\_fiq中。

为了实现来自同一中断源的中断嵌套，相应的R14和SPSR中的内容必须被保存到堆栈中。这种做法可以很容易地由程序指令使用R13作为堆栈指针来实现。所以，在所有模式中都有专用的寄存器R13和R14。中断服务程序可以将R14和SPSR保存到自己的私有堆栈中，然后清除CPSR中的中断屏蔽位。也可以将其他寄存器保存到私有堆栈中以产生所需要的额外操作空间。FIQ模式用作快速中断，可以利用从R8\_fiq到R13\_fiq的专用寄存器，所以不需要将寄存器保存到堆栈中。

我们在第3章中曾指出LDM和STM指令能够传送多个字，可以非常方便地用于处理堆栈操作。例如，使用R13作为堆栈指针，子程序或中断服务程序可以使用下面的指令来保存寄存器和返回地址：

228

```
STMFDF R13!, { R0, R1, R2, R14 }
```

相似地，LDMFDF指令可以用来恢复被保存的值。在SWI或指令预取异常中，恢复到R14中的值也是正确的返回地址。因此，这个值可以直接恢复到R15中，返回到被中断程序的指令如下：

```
LDMFDF R13!, {R0, R1, R2, R15}^
```

指令末尾的“^”符号与前面SUBS指令中S后缀的作用相同，它使处理器在装入R15的同时将SPSR复制到CPSR中。需要注意的是LDM不能用来从IRQ和FIQ中断返回，因为R14中的内容必须首先进行如表4-2所示的修正。

**例4.4** 图4-13给出了一个使用中断的例子，它是图4-9中的程序对ARM处理器的重写。我们已经假设键盘是连接到中断线IRQ上的，并且在相应的中断向量位置包含一条跳转到READ的指令。还假设在进入这个代码段时存储器缓冲区地址LINE已经装到了PNTR单元中。同时假定PNTR和EOL的单元在地址空间里是足够接近的，它们可以使用相对地址模式进行访问。主程序通过设置键盘CONTROL寄存器中的KEN标志并清除处理器状态寄存器的I屏蔽位，来允许键盘接口和处理器中断。清除I屏蔽位（第7位）是通过MSR指令将值\$50装入到CPSR中来实现的。

在中断服务程序中，我们使用LDM和STM指令保存和恢复寄存器，使用SUBS指令返回被中断的程序，使用3.4.1节中描述的ADR指令将图4-3中键盘的DATAIN寄存器的地址装入到处理器寄存器中。假定控制寄存器CONTROL的地址为DATAIN+3。

主程序			
MOV	R0,#0		
STR	R0,EOL	清除EOL标志	
ADR	R1,DATIN	载入寄存器DATIN的地址	
LDRB	R0,[R1,#3]	获取CONTROL寄存器的内容	
ORR	R0,R0,#4	设置寄存器CONTROL中的KEN位	
STRB	R0,[R1,#3]	来允许键盘中断	
MOV	R0,#&50	在处理器中允许IRQ中断并转换到	
MSR	CPSR,R0	用户模式	
:			
IRQ中断服务程序			
READ	STMTFD	R13!,{R0-R2,R14_irq}	保存R0、R1和R14_irq到堆栈中
	ADR	R1,DATIN	载入寄存器DATIN的地址
	LDRB	R0,[R1]	获取输入字符
	LDR	R2,PNTR	载入指针值
	STRB	R0,[R2],#1	存储字符并增大指针
	STR	R2,PNTR	更新存储器中指针的值
	CMPB	R0,#&0D	检查是否为回车符
	LDMNEFD	R13!,{R0-R2,R14_irq}	如果不是,恢复寄存器并返回
	SUBNES	PC,R14_irq,#4	
	LDRB	R0,[R1,#3]	否则,获取CONTROL寄存器
	AND	R0,R0,#&FB	消除KEN位来禁止键盘中断
	STRB	R0,[R1,#3]	
	MOV	R0,#1	设置EOL标志
	STR	R0,EOL	
	LDMFD	R13!,{R0-R2,R14 }	恢复寄存器
	SUBS	PC,R14_irq,#4	并返回

图4-13 对应图4-9,从键盘读入一行输入的ARM中断服务程序

### 4.3.2 68000中断结构

68000有8个中断优先级。在特定时间处理器运行的优先级被编码在处理器状态字的3个位中,如图4-14所示,其中0级的优先级最低。与68000相连的I/O设备采用与图4-8b相似的排列方式,在这种方式中中断请求被分配给1到7范围内的优先级。中断请求只有在它的优先级高于处理器的优先级时才被接受,但有一个例外:优先级为7的请求在任何时候都会被接受,它是一个边缘触发的不可屏蔽的中断。处理器接受一个中断请求后,在执行中断服务程序之前PS寄存器中指示的优先级自动提升为中断请求的优先级。因此,除了7级以外,其他同级或低级的中断请求将被禁止,7级中断总是被允许的。

在中断发生时处理器自动保存程序计数器和处理器状态字的内容。PC被压入到处理器的堆栈中,紧接着PS也被压入到堆栈中,使用寄存器A7作堆栈的指针。在68000汇编语言中,中断返回指令称为异常返回(RTE),将栈顶单元弹出到PS中,将下一个单元弹出到PC中。如图4-14所示,PS寄存器中包括一个管理位S和一个跟踪位T。S位确定处理器运行在管态模式(S=1)还是用户模式(S=0)。T位允许一类特殊的中断,称为跟踪异常,在4.2.4节中已经进行了描述。这些信息在接受中断时将被自动保存起来,在中断服务结束后重新恢复。而任何其他需要保存的附加信息,如通用寄存器的内容,必须在中断服务程序内部被明确地保存和恢复。

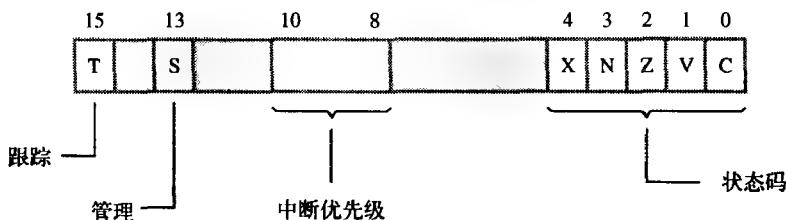


图4-14 68000处理器的处理器状态寄存器

68000处理器使用了向量中断。当接受一个中断请求时，从存储在主存中的中断向量中获取中断服务程序的地址。68000共有256个中断向量，编号为0到255。每个中断向量由32位组成，这32位构成所需要的起始地址。当一台设备请求中断时，它通过向处理器发送指向相应中断向量的8位向量编号来响应中断应答信号。此外，68000还提供了一个自动向量功能，它取代了发送向量编号。设备可以激活一条专门的总线控制线路表示它希望使用自动向量功能。在这种情况下，处理器将根据中断请求的优先级选择7个专门用于此目的的向量中的一个。

**例4.5** 图4-15是一个在68000中使用中断的例子。它是图4-9中的程序对68000的重写。我们假定键盘接口电路使用自动向量，并且产生的中断请求的优先级为2。因此，为了允许中断，处理器的优先级必须设置成低于2级。当将位匹配模式\$100装入到寄存器PS后，处理器的优先级就被设置为1。

主程序			
	MOVE.L	#LINE,PNTR	初始化缓冲区指针
	CLR	EOL	清除行末指示器
	ORL.B	#4,CONTROL	设置KEN位
	MOVE	#\$100,SR	设置处理器优先级为1
	...		
中断服务程序			
READ	MOVEM.L	A0/D0, -(A7)	保存寄存器A0, D0到堆栈
	MOVEA.L	PNTR,A0	载入地址指针
	MOVE.B	DATAIN,D0	获取输入字符
	MOVE.B	D0,(A0)+	将它存储到存储器缓冲区
	MOVE.L	A0,PNTR	更新指针
	CMPL.B	#\$0D,D0	检查是否为回车符
	BNE	RTRN	
	MOVE	#1,EOL	指示行末
	ANDL.B	#\$FB,CONTROL	清除KEN位
RTRN	MOVEM.L	(A7)+,A0/D0	恢复寄存器D0, A0
	RTE		

图4-15 对应图4-9，从键盘读入一行输入的68000中断服务程序

### 4.3.3 Pentium的中断结构

Pentium处理器是IA-32体系结构的一个例子。这种体系结构使用两根中断请求线：一根是不可屏蔽中断（NMI），另一根是可屏蔽中断，也称为用户中断请求INTR。NMI上的中断请求任何时候都必须被处理器接受。而INTR上的中断请求只有在它的优先级高于当前执行程序的优先级

时才被接受，后面我们将对此简要介绍。也可以通过设置处理器状态寄存器中的中断允许位来允许和禁止INTR中断的发生。

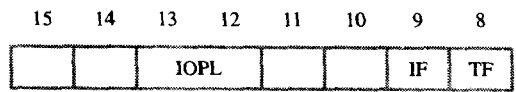
除了外部中断外，在程序执行期间发生的事件可以导致异常。这些事件包括无效操作码、除法错误、溢出以及其他事件，也包括跟踪和断点中断。

231

这些事件中任何一个的发生都会使得处理器转移到中断服务程序中。每一个中断或异常都分配一个向量编号。在INTR中断中，当中断请求被响应时，向量编号由I/O设备通过总线发送。对于其他异常，向量编号都是预先分配的。根据向量编号，处理器从中断向量表中获得中断服务程序的起始地址。

Pentium处理器还附带有高级可编程中断控制器（APIC）芯片。各种设备通过这个芯片连接到处理器。中断控制器在不同设备间实现一个优先级结构，并为每一台设备发送一个适当的向量编号给处理器。

图3-37中显示了Pentium处理器的状态寄存器，在Intel文献中称为EFLAGS。图4-16显示的是这个寄存器的第8位到第15位，包括中断允许标志（IF）、陷阱标志（TF）和I/O优先级（IOPL）。当IF=1时，INTR中断将被接受。陷阱标志允许在每一条指令后产生跟踪中断。



232

图4-16 Pentium处理器状态寄存器的一部分

Pentium处理器的优先级结构非常复杂，操作系统的不同部分在四个优先级下分别执行。四个优先级分别占用处理器地址空间的不同区段。从一个优先级转换到另一个优先级需要经过门机制的一系列检查。这样构建的OS是高度安全的。此外，处理器也可以在一个简单的模式下运行，在这个模式中没有优先级，所有程序运行在同一区域中。在这里我们只讨论这种简单的模式。

当接收到一个中断请求或发生一个异常时，处理器将执行以下操作：

1. 将处理器状态寄存器、当前段寄存器（CS）和指令指针（EIP）压入到处理器堆栈中，处理器堆栈指针（ESP）指向该堆栈。
2. 如果异常是由不正常的执行状态引起的，处理器还要将描述异常原因的代码压入到堆栈中。
3. 需要的话，处理器将清除相应的中断允许标志以禁止来自同一中断源的更多中断。
4. 根据该中断的向量编号从中断向量表中取出中断服务程序的起始地址，并将它装入到EIP中，然后继续执行。

在处理完中断请求后，例如传送输入或输出数据，中断服务程序使用中断返回指令IRET返回被中断的程序。该指令将EIP、CS和处理器状态寄存器从堆栈中弹出到相应的寄存器中，恢复处理器的状态。

在子程序情况下，中断服务程序可以创建临时工作空间，用来保存寄存器或对局部变量使用的堆栈结构。但它必须在执行IRET指令之前恢复所有被保存的寄存器，并确保堆栈指针ESP是指向返回地址的。

**例4.6** 图4-17显示的是图4-9中程序对Pentium的重写。假定键盘发送的中断请求向量编号为32，并且中断服务程序的起始地址READ已经装入到中断向量表相应的入口地址处。使用STI指令将处理器状态寄存器中的IF标志位置1，使处理器允许中断。

233

<b>主程序</b>		
	MOV	EOL,0
	MOV	BL,4
	OR	CONTROL,BL
	STI	
	...	
<b>中断服务程序</b>		
READ	PUSH	EAX
	PUSH	EBX
	MOV	EAX,PNTR
	MOV	BL,DATAIN
	MOV	[EAX],BL
	INC	DWORDPTR[EAX]
	CMP	BL,0DH
	JNE	RTRN
	MOV	BL,4
	XOR	CONTROL,BL
	MOV	EOL,1
RTRN	POP	EBX
	POP	EAX
	IRET	

图4-17 在IA-32处理器中，从键盘读入一行输入的中断服务程序

#### 4.4 直接存储器访问

在前面的章节中我们主要讲述的是处理器和I/O设备之间的数据传输。这些数据是通过执行指令完成传输的，如指令：

Move DATAIN, R0

传递输入或输出数据的指令只有在处理器确定I/O设备准备就绪时才能执行。处理器通过查询设备接口的状态标志，或等待设备发送中断请求来确定设备是否就绪。但是这两种方式都会导致相当大的开销，因为每传递一个字大小的数据必须执行好几条程序指令。除了查询设备的状态寄存器以外，还需要使用指令来增加存储器地址和记录字数。当使用中断时，保存和恢复程序计数器和其他状态信息也会产生额外的开销。

**234** 除了查询设备的状态寄存器以外，还需要使用指令来增加存储器地址和记录字数。当使用中断时，保存和恢复程序计数器和其他状态信息也会产生额外的开销。

为了高速传输大量的数据，可采用另一种可行的方法。该方法提供了一个专门的控制单元，使一块数据可以直接在外部设备和主存之间传输，而不需要处理器做连续干预。这种方法称为直接存储器访问（DMA）。

DMA传输由I/O设备接口中的控制电路来执行。我们将这种电路称为DMA控制器。访问主存时，DMA控制器执行一般由处理器执行的功能。对每一个传输的字，它提供存储器地址和所有控制数据传输的总线信号。因为必须传输大量的数据，DMA控制器必须为连续的字增加存储器地址并累计已传输的字数。

虽然DMA控制器不需要处理器的参与就可以传输数据，但它的操作必须由处理器执行的程序所控制。为了初始化一个字块的传输，处理器必须向它发送起始地址、块内字数和传输方向等信息。接收到这些信息后，DMA控制器就开始执行所请求的操作。当整块数据传输完成后，控制器产生一个中断信号通知处理器。

在DMA执行传输时，请求传输的程序被中断，处理器可以执行其他的程序。DMA传输完成后，处理器可以返回到请求这次传输的程序中。

I/O操作总是在计算机操作系统响应应用程序的请求时才被执行的。而且，OS也负责管理中断正在执行的程序和启动另一个程序。因此，对于一次包含DMA的I/O操作，OS首先将请求传输的程序设置为阻塞状态（见4.2.6节），启动DMA操作，然后开始执行另一程序。当传输完成后，DMA控制器发送一个中断请求通知处理器。然后OS进行响应，将被中断的程序置为就绪状态，使得它可以被调度程序选定继续执行。

图4-18显示了一个DMA控制器的寄存器，处理器访问该寄存器来初始化传输操作。其中两个存储器用来存储起始地址和字数。第三个寄存器保存着状态和控制标志。R/ $\bar{W}$ 位决定传输的方向。当它被程序指令置1时，控制器执行读操作，即将数据从存储器传输到I/O设备。否则，控制器执行写操作。当控制器传输完一块数据并准备好接收另一个命令时，将Done标志置1。第30位是中断允许标志IE。当这个标志被置1时，控制器在传输完一块数据后将产生一个中断。最后，在请求中断后控制器将IRQ位置1。

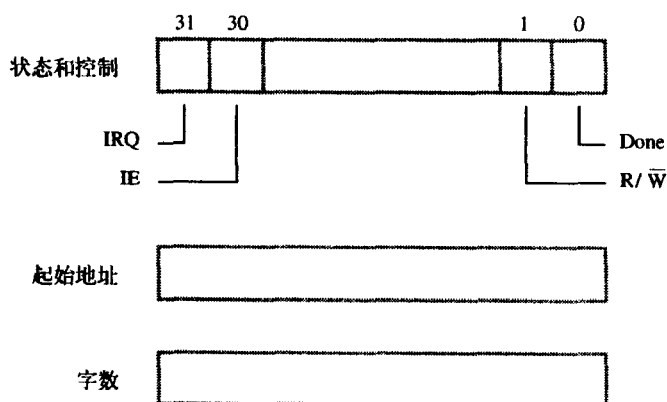


图4-18 DMA接口中的寄存器

图4-19是一个计算机系统的例子，它显示了如何使用DMA控制器。其中一个DMA控制器将高速网络连接到计算机总线上。磁盘控制器控制两个磁盘，同时具有DMA功能，提供两个DMA通道。它能够执行两个独立的DMA操作，就像每个磁盘都有自己的DMA控制器一样。它有双份寄存器来存放存储器地址、字数等信息，因此每台设备可以使用一套。

要启动一次DMA传输，将一块数据从主存传送到其中一个磁盘，程序需要将地址和字数信息写入到磁盘控制器相应通道的寄存器中，还要向磁盘控制器提供将来检索要用到的识别数据。DMA控制器独立执行指定的操作。当DMA传输完成后，通过设置Done位在DMA通道的状态和控制寄存器中记录该事件。同时，如果IE位被置位，控制器将发送一个中断请求到处理器并设置IRQ位。状态寄存器也可以用来记录其他信息，比如传输是正确的还是错误的。

处理器和DMA对存储器的访问可以交织进行。在这种方式中，DMA设备请求使用总线的优先级要高于处理器。在不同的DMA设备中，最高优先级分配给了高速外围设备，如磁盘、高速网络接口或图形显示设备。因为处理器占用了存储器大部分的访问周期，所以认为DMA控制器“窃取”了处理器的存储器访问周期。因此，这种交织技术通常称为周期窃取（cycle stealing）技术。除这种方式外，还可以给DMA控制器一个专门的访问主存周期，使它可以不被中断地传输一块数据。这种方式称为块或脉冲模式。



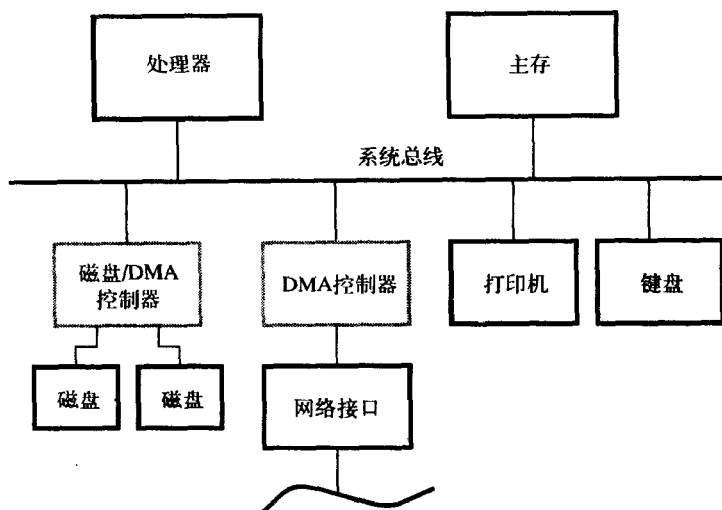


图4-19 DMA控制器在计算机系统中的使用

大部分DMA控制器中都有一个数据存储缓冲区。例如，在图4-19的网络接口中，DMA控制器从主存中读出一块数据并存储到它的输入缓冲区中。该传输使用脉冲模式，传输的速度既适合于存储器也适合于计算机总线。因此，缓冲区的数据用网络速度在网络上发送。

如果处理器和一个DMA控制器或两个DMA控制器同时都要使用总线访问主存，就会产生冲突。为了解决这些冲突，在总线上提供有一个仲裁程序来协调所有请求存储器传送的设备的活动。

### 总线仲裁

在任何时候被允许在总线上启动数据传输的设备称为总线主控制器。在当前主控制器放弃对总线控制时，另一设备可以获得对总线的控制权。总线仲裁就是选择下一个成为总线主控制器的设备并将主控权传递给它的进程。总线主控制器的选择必须考虑各种设备的需求，这可以通过建立一个访问总线的优先级系统来实现。

现有的总线仲裁方法有两种：集中式和分布式。在集中式仲裁中，只有单个总线仲裁器执行仲裁。在分布式仲裁中，所有设备都参与下一个总线主控制器的选择。

#### 集中式仲裁

在这种方式中，总线仲裁器可以是处理器或一个连接到总线上的独立单元。图4-20显示了一种基本结构，总线仲裁电路包含在处理器中。在这种结构中，处理器通常是总线主控制器，除非它将总线主控权授予其中一个DMA控制器。DMA控制器通过激活总线请求线  $\overline{BR}$ ，来表示它需要成为总线主控制器。这条线是漏极开路式的，和图4-6中中断请求线原理相同。总线请求线上的信号是所有与之相连设备的总线请求的逻辑“或”。当总线请求信号被激活时，处理器激活总线允许信号  $BG1$ ，指示DMA控制器当总线空闲时它可以使用总线。这个信号采用菊花链结构连接到所有DMA控制器上。因此，如果DMA控制器1正在请求总线，它就阻塞这个允许信号的传播。否则，它启动信号  $BG2$ ，将允许信号传递给后面的设备。当前总线主控制器通过激活另一条集电极开路线  $\overline{BBSY}$ ，向所有设备表示它正在使用总线。因此，在收到总线允许信号后，DMA控制器等待  $\overline{BBSY}$  信号变为无效，然后获得总线主控权。同时，它又激活  $\overline{BBSY}$  信号阻塞其他设备使用总线。

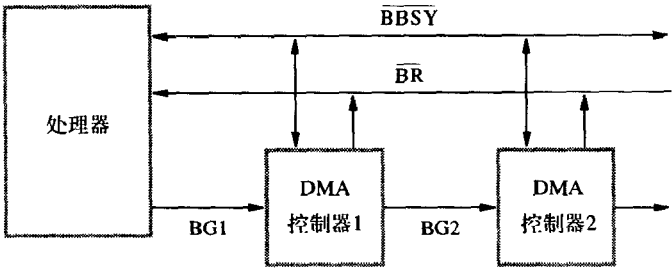


图4-20 一种使用菊花链的简单总线仲裁结构

图4-21中的时序图显示了图4-20中的设备在DMA控制器2请求总线、获得总线主控权到最后释放总线的过程中发生的事件顺序。在DMA控制器2作为总线主控器期间，它可以进行一次或多

次数据传送操作，这取决于它是工作在周期窃取模式还是脉冲模式。当它释放总线后，处理器重新获得总线主控权。这个图显示的只是仲裁过程中信号间的先后因果关系。至于具体的定时，因

238

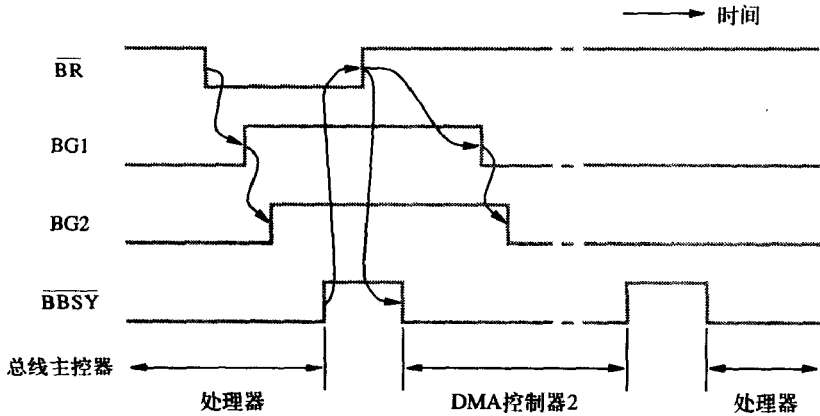


图4-21 图4-20中的设备在总线主控权传递过程中的信号发生顺序

239

图4-20显示的是一条总线请求线和一条总线允许线形成的菊花链。在相似的图4-8b的多中断请求结构中提供了多个这样的线对。这种结构使得确定不同设备的请求顺序相当复杂。仲裁电路根据预先确定的优先级确保在任何时候都只有一个请求被允许。例如，如果有四条总线请求线，BR1到BR4，就可以使用一种固定优先级方式，BR1被授予最高的优先级，BR4被授予最低的优先级。此外，也可以使用轮转优先级方式使所有设备都有均等的机会。所谓轮转优先级，就是说当BR1线上的请求被允许后，优先级顺序就变成了2, 3, 4, 1。

分布式仲裁

分布式仲裁指的是在仲裁过程中不使用中心仲裁器，所有等待使用总线的设备有着同等的机会。图4-22是一种简单的分布式仲裁方案。总线上的每一台设备都分配有一个4位识别码。当一台或多台设备请求总线时，它们启动 Start - Arbitration 信号并将它们的4位ID号码放到四条集电极开路线 ARB0 到 ARB3 上。所有竞争者发送到这些线路上的信号之间相互发生作用，最后的胜出者被选定。最终这四条线组成的代码代表有最高ID号码的请求。

线路的驱动器是集电极开路类型的。因此，如果一个驱动器的输入等于1，另一个连接到

同一总线的驱动器的输入等于0，则总线将处于低电平状态。也就是说，是按逻辑“或”进行连接的。

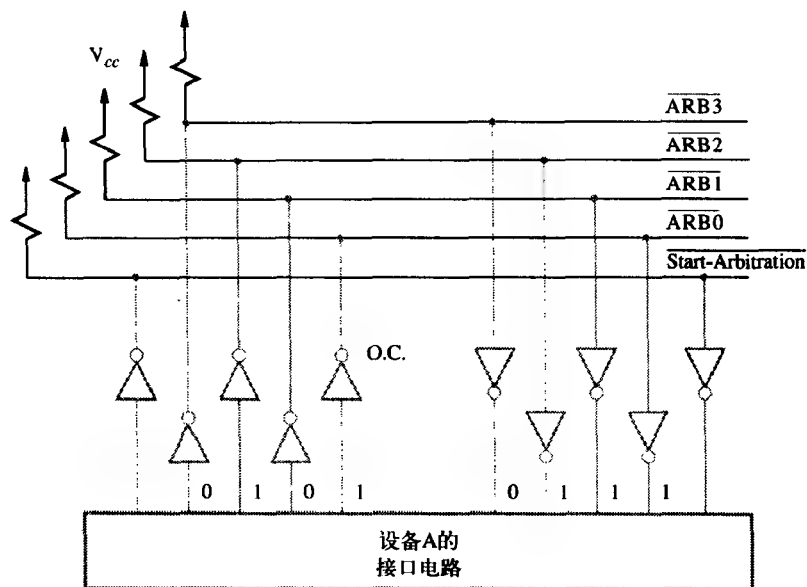


图4-22 一种分布式仲裁方案

假设有两台设备A和B正在请求使用总线，ID号码分别为5和6。设备A发送位序0101，设备B发送位序0110。最终“或”的结果是0111。每台设备从高位开始和仲裁线上的值比较。如果它在某一位检测到不一致，就在这一位以及所有后继位关闭驱动器，即将这些驱动器的输入置0。在该例中，设备A在 $\overline{ARB1}$ 上检测到不一致，因此禁止 $\overline{ARB1}$ 和 $\overline{ARB0}$ 的驱动器，从而导致仲裁线上的位序变为0110，这意味着B在竞争中获得了胜利。需要注意的是，因为优先级线上的代码在一个较短的时间内仍是0111，所以设备B可能暂时在 $\overline{ARB0}$ 线上关闭它的驱动器。但是A的操作将会导致 $\overline{ARB1}$ 线变成0，一旦B见到 $\overline{ARB1}$ 线为0，它将重新启动驱动器。

分布式仲裁具有较高的可靠性，因为这种总线的操作不依赖于任何一台单独的设备。目前，还有很多其他方案用来实现分布式仲裁。4.7.2节中将要描述的SCSI总线就是分布式仲裁的一个例子。

## 4.5 总线

处理器、主存和I/O设备可以通过公共总线相互连接。公共总线的主要职能是为数据传输提供通信通道。它有支持中断和仲裁所需要的线路。在本节中，我们讲述用于数据传输的总线协议的主要特性。总线协议是管理连接到总线上各设备的行为规则的集合，如何时将数据放到总线上、启动控制信号等。描述完总线协议后，将给出一些使用这种协议的接口电路的例子。

用来传输数据的总线线路可以分为三种类型：数据线、地址线和控制线。控制信号说明是否执行行读或写操作。通常，使用单根 $R/\overline{W}$ 线来传送控制信号，当它被置1时表示进行读操作，被置0时表示进行写操作。在允许传输不同长度的操作数时，如字节、字或长字，数据的长度也被显示出来。

总线控制信号还要传送时序信息。这些信息详细说明何时处理器和I/O设备可以将数据放到总线上或从总线上接收数据。目前有多种传输数据时序的方式，大致可以将它们分为同步和异步两种方式。

回顾4.4.1节，我们知道在任何数据传输操作中都有一台设备作为主控制器。它在总线上发布读写命令、启动数据传输，因此也可以称作启动设备。通常，处理器是总线主控制器，但其他有DMA功能的设备也可以成为总线主控制器。被主控设备寻址的设备称作从动或目标设备。

240

4.5.1 同步总线

在同步总线中，所有设备都从一根公共时钟线路上获取时序信息。这条线路上相同间隔的脉冲定义了相同的时间间隔。在最简单的同步总线中，每一个时间间隔组成一个总线周期，一个周期内可以进行一次数据传送。图4-23举例说明了这种方式。在该图及后续图中，地址和数据线同时显示出了高电平和低电平。用高或低电平表示是一种通用的约定，它依赖于具体的地址和数据传输模式。交点表示的是发生了变化。处于不确定或高阻抗状态的单信号线路用高低电平之间的中间电平来表示。

让我们来看一下在一次输入（读）操作期间发生事件的顺序。在时间  $t_0$  时，总线主控制器将设备地址放到地址线上并向控制线发送一条相应的命令。如需要，这个命令还要说明这是一次输入操作并指明将要读取操作数的长度。数据在总线上的传输速度取决于总线的物理和电气特性。时钟脉冲的宽度  $t_1 \sim t_0$  必须大于两台连接到总线上的设备之间的最大传播延迟。而且，该时间要足够长来保证所有设备对地址和控制信号进行译码，从而使得被寻址设备（从动设备）能够在时间  $t_1$  进行响应。此外，确保在  $t_1$  前从动设备不执行任何操作也不将数据放到总线上同样十分重要。因为总线上的信息在  $t_0$  到  $t_1$  期间是不可靠的，此时信号正处于改变状态。被寻址的从动设备应在时间  $t_1$  时将请求的数据放到数据总线上。

时钟周期结束时，即在时间  $t_2$ ，主控设备选通数据线上的数据存入到自己的输入缓冲区中。在这里，“选通”指的是在特定时刻捕捉数据的值并将它们存到缓冲区中。要使数据能够正确地装入到存储设备中（如触发寄存器），数据在存储设备输入端的有效时间必须大于存储设备的准备时间（见附录A）。因此， $t_2 \sim t_1$  这段时间必须大于总线上的最大传输时间与主控设备输入缓冲寄存器的准备时间之和。

输出操作也有类似的过程。主控设备在发送地址和命令信息的同时也将输出数据放置到数据线上。在时间  $t_2$ ，被寻址的设备选通数据线，将数据装入到自己的数据缓冲区中。

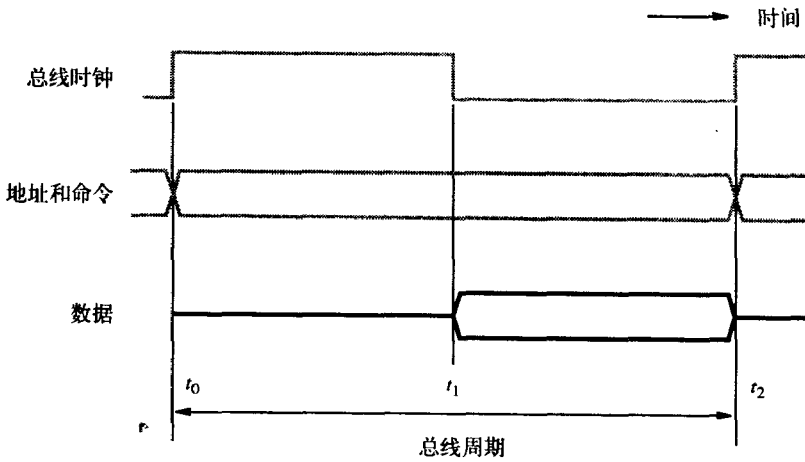


图4-23 同步总线上的输入传输时序

241

图4-23中的时序图是总线线路上发生动作的一种理想化表示。实际中信号变换状态的确切的时间与图中显示的有些差别,因为在总线线路和设备电路上存在着传输延迟。图4-24给出了一个更接近实际情况的图。在这幅图中除了时钟以外,每个信号都显示了两个图像。因为信号要花费时间从一台设备传送到另一台设备,所以不同的设备见到同一信号跳变的时间不同。其中一个显示的是主控设备见到的信号,另一个显示的是从动设备见到的信号。我们假设总线上所有设备见到时钟变换的时间相同。系统设计者需要花费相当大的精力来确保时钟信号能满足这一条件。

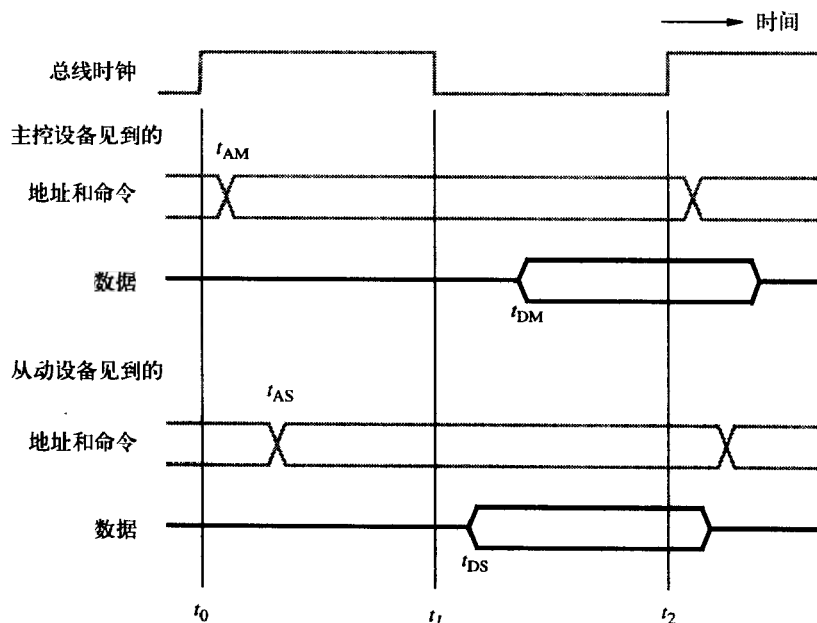


图4-24 对应图4-23,更实际的输入传输时序图

主控设备在时钟周期1开始的上升沿( $t_0$ )发送地址和命令信号。但由于总线驱动电路的延迟,在 $t_{AM}$ 之前,这些信号并不出现在总线上。在 $t_{AS}$ 时刻,信号到达从动设备。从动设备对地址信号进行译码,并在时间 $t_1$ 发送被请求的数据。同样,数据信号直到 $t_{DS}$ 时刻才出现在总线上。它们向主控设备方向传输,在 $t_{DM}$ 到达。在时间 $t_2$ ,主控设备将数据装入到它的输入缓冲区中。因此, $t_2 \sim t_{DM}$ 这段时间是主控设备输入缓冲区的准备时间。数据在 $t_2$ 后必须保持有效一段时间,这段时间等于该缓冲区的保持时间。

时序图在文献中常常只给出图4-23中的简化图,尤其是在概括介绍数据是如何传输时。但实际上,信号总是包含有图4-24所示的延迟。

### 多周期传输

按照上面描述的方式我们可以得到设备接口的一种简单设计方式。但是,这种方式有它的局限性。因为每次传输必须在一个周期内完成,时钟周期 $t_2 \sim t_0$ 必须满足最长的时间延迟和最慢的设备接口。这将迫使所有设备都在最慢设备的速度下工作。

此外,处理器也不能确定被寻址设备是否已经响应。它只是简单假定在 $t_2$ 时输出数据已经被I/O设备接收,或输入数据已在数据线上有效。但如果由于故障,设备没有响应,则无法检测到该错误。

为克服这些局限性,大部分总线都有表示设备响应的控制信号。这些信号通知主控设备从动

设备已经识别了它的地址并准备好参与数据传输操作了。也可以调整数据传输时的脉冲宽度来适应设备。为简化这一过程，我们使用高频时钟信号，这样一个完整的数据传送周期就可能跨越好几个时钟周期。因此，不同设备传输数据所需的时钟周期数就有可能不相同。

图4-25是多时钟周期传输的一个例子。在时钟周期1期间，主控设备将地址和命令信息发送到总线上请求一次读操作。从动设备接收到这个信息并将其译码。在下一个时钟的上升沿，即时钟周期2开始时，决定进行响应并开始访问被请求的数据。我们已经假设在获取数据的过程中存在一些延迟，所以从动设备并不能立即响应。在时钟周期3数据被准备好并放到总线上。同时，从动设备启动从动就绪控制信号。等待这个信号的主控设备在时钟周期3结束时选通数据到它的输入缓冲区中。至此总线传送操作完成，主控设备可以在时钟周期4发送一个新的地址并启动下一次新的传送。

从动就绪信号是从动设备对主控设备的应答，它确认有效数据已经被发送了。在图4-25中，从动设备在周期3响应。其他设备响应的时间可能会早一些或晚一些。从动就绪信号使得不同设备的总线传输脉冲宽度可以不同。如果被访问的设备根本不响应，主控设备等待预先确定的最大时钟周期数后就放弃这次操作。这可能是由于地址错误或设备故障导致的。

243

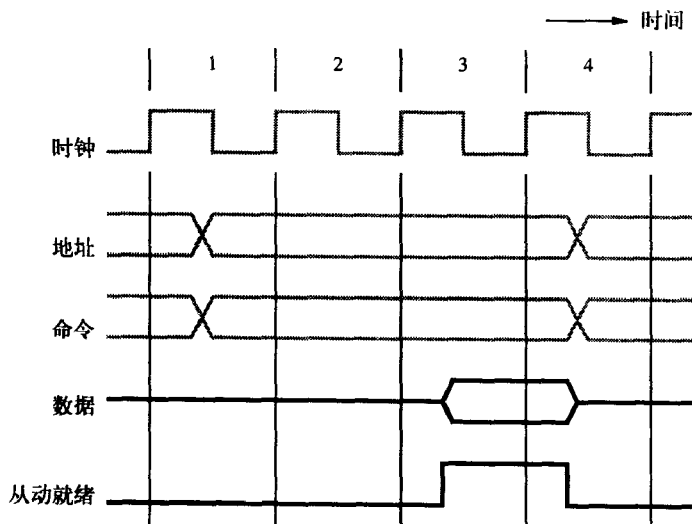


图4-25 多时钟周期的输入传输

需要注意的是计算机总线上使用的时钟信号并不需要与处理器的时钟相同。后者通常要快得多，因为它是控制处理器芯片的内部操作的。例如，将信号输入到芯片中遇到的延迟要远小于在连接印刷电路板上芯片的总线时遇到的延迟。时钟频率是高度依赖于技术的，在现代的处理器芯片中，一般都超过500MHz。而在存储器和I/O总线中，时钟频率可能在50到150 MHz的范围内。

许多计算机总线，如Pentium和ARM处理器总线，使用与图4-25相似的方式控制数据传输。4.7.1节将要描述的PCI总线标准也与图4-25十分相似。接下来我们将讲述一种根本不使用时钟信号的方法。

#### 4.5.2 异步总线

控制总线上数据传输的另一种方式是基于主控设备和从动设备之间的握手信号。“握手”的概

念是图4-25中从动就绪信号概念的泛化。公共时钟被两根定时控制线取代,分别是主控就绪和从动就绪。第一根控制线由主控设备启动表示它已经准备好一次事务,第二根则是从动设备的响应。

原理上,握手协议控制的数据传送按照如下步骤进行。主控设备将地址和命令信息放到总线上。然后,激活主控就绪线通知所有设备已经将地址和命令信息放到总线上了。总线上所有设备对该地址进行译码。被选定的设备执行请求操作,并激活从动就绪线通知处理器它已经准备好。主控设备等待从动就绪线有效后,将自己的信号从总线上撤销。在读操作中,还需要将选通数据存入到输入缓冲区中。

图4-26是一个基于握手方式的输入数据传输操作的时序,它描述了如下的事件顺序:

$t_0$ ——主控设备将地址和命令信息放到总线上,总线上所有设备开始对该信息进行译码。

$t_1$ ——主控设备将主控就绪线置1,通知所有I/O设备地址和命令信息已经放到总线上了。 $t_1 \sim t_0$ 的延迟用来补偿总线上出现的相位偏移。当同一信号源同时发送的两个信号不同时到达目的地时,就会出现相位偏移。这是由总线中不同线路的传播速度不同导致的。因此,为了保证主控就绪信号不在地址和命令信息之前到达设备,延迟 $t_1 \sim t_0$ 应该大于最大的总线相位偏移。(注意:在同步情形中,总线相位偏移是最大传播延迟的一部分。)当地址信息到达设备时,接口电路都对它们进行译码。要保证有足够的时间进行地址译码,该延迟也应该包含在 $t_1 \sim t_0$ 内。

$t_2$ ——被选定的从动设备在对地址和命令信息译码后,将它数据寄存器中的数据放到数据线上执行请求的输入操作。同时,将从动就绪信号置1。如果在数据放到总线之前接口电路产生了额外延迟,从动设备必须相应地延迟从动就绪信号。 $t_2 \sim t_1$ 这段时间的大小依赖于主控设备和从动设备之间的距离以及从动设备电路产生的延迟。总线异步的特性就体现在该段延迟时间的可变性上。

$t_3$ ——从动就绪信号到达主控设备,表示输入数据已经在总线上可用。不过,因为我们假定设备接口在发送从动就绪信号的同时将数据放置到总线上,所以主控设备应该允许总线相位偏移。此外,还必须允许输入缓冲区所需的准备时间。在延迟最大总线相位偏移和最小准备时间后,主控设备选通数据到它的输入缓冲区,同时撤销主控就绪信号,表示它已经收到数据。

$t_4$ ——主控设备从总线上撤销地址和命令信息。 $t_3$ 与 $t_4$ 之间的延迟也是用来补偿总线相位偏移的。因为如果设备在总线上见到的地址发生变化,主控就绪信号仍然等于1就会产生错误的寻址。

$t_5$ ——当设备接口收到主控就绪信号从1到0的跳变后,就从总线上撤销数据和从动就绪信号。此时输入传输结束。

图4-27显示的是输出操作的时序,实质上与输入操作相同。在输出操作中,主控设备在将输出数据放置到数据线的同时发送地址和命令信息。选定的从动设备收到主控就绪信号后选通数据到它的输出缓冲区,并将从动就绪信号置1表示它已经接收到数据了。这个周期的其余部分与输入操作相同。

在图4-26和图4-27的时序图中,我们假设主控设备将补偿总线相位偏移和地址译码延迟,引入了 $t_0$ 至 $t_1$ 段和 $t_3$ 至 $t_4$ 段的延迟。如果这些延迟能够提供足够的时间给I/O接口译码地址,接口电路就可以直接用主控就绪信号来控制发送给或来自总线的信号,这一点在学习下一节的接口电路示例时会更加清楚。

图4-26和图4-27中的握手信号是完全互锁的。一个信号状态的改变将会导致另一个信号状态的改变。因此这种方式称为完全握手方式。它提供了最高程度的灵活性和可靠性。

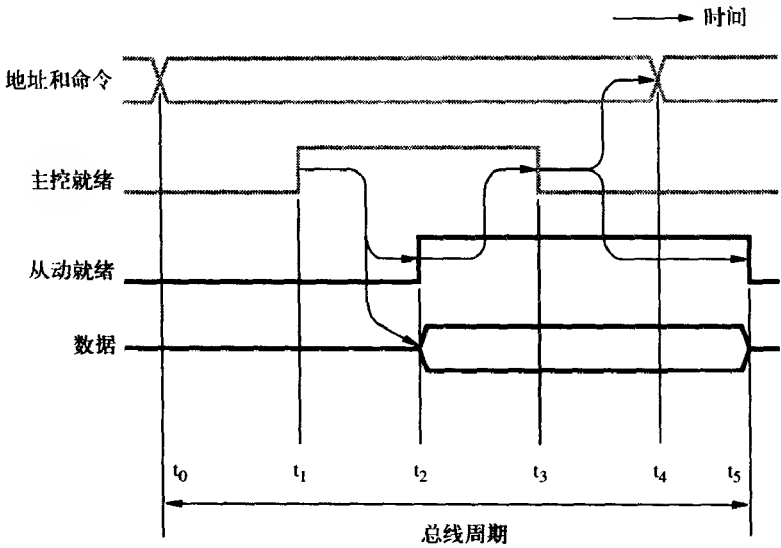


图4-26 输入操作中数据传输的握手控制

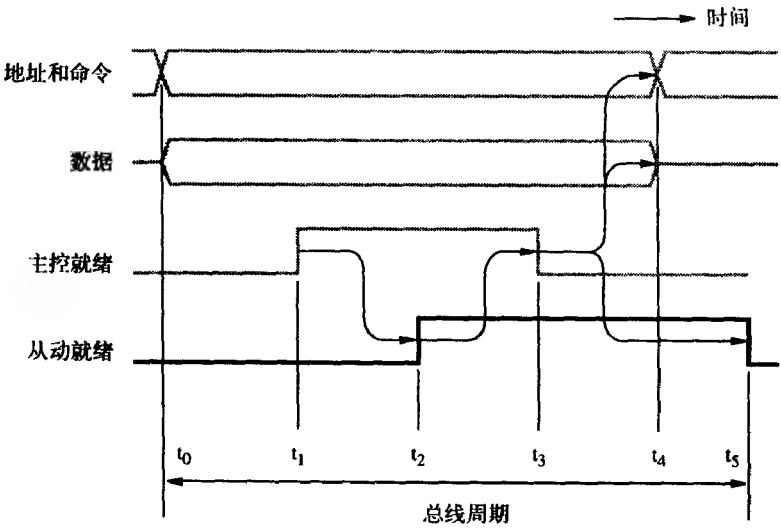


图4-27 输出操作中数据传输的握手控制

### 4.5.3 讨论

在商业计算机中使用了许多上述类似的总线技术。例如，68000系列处理器中的总线就有两种工作模式——同步和异步。具体设计的选择需要权衡如下因素：

- 接口电路的简单性。
- 适应不同延迟的设备接口的能力。
- 总线传输需要的总时间。
- 检测由于寻址不存在的设备或接口故障所导致的错误的能力。

异步总线的主要优点是握手过程中不需要同步发送者和接收者的时钟，从而简化了时序设计。接口电路或总线线路上传播过程中产生的延迟都可以很容易解决。当这些延迟改变时，例如在增



加或删除一个接口电路时由于负载的变化导致的延迟变化，数据传输的时序可以自动根据新情况进行调整。而对于同步总线，时钟电路必须仔细设计以确保正确的同步，并且延迟也必须严格控制在一定范围之内。

由完全握手方式控制的异步总线上的数据传送，由于每一次传送都要包括两个来回的延迟（四个端到端的延迟），所以传输速率受到很大的限制。从图4-26和图4-27中我们可以很容易看出，从动就绪信号必须等到主控就绪信号的跳变到达后才能跳变，反之亦然。在同步总线中，时钟周期只需满足一个端到端的传播延迟即可。因此，同步总线可以得到更高的传输速率。如前面所述，为了适应较慢的设备还使用了附加的时钟周期。今天我们使用大部分的高速总线都是使用同步方法。

247

## 4.6 接口电路

接口是由将I/O设备连接到计算机总线上的电路组成的。在接口的一侧有总线地址信号、数据信号和控制信号。在另一侧有一条数据通路，该通路和它相关的控制器一起负责接口与I/O设备之间的数据传输，这一侧称为端口，可以分成并行和串行两类。并行端口每次可以同时传输多位数据，一般为8位或16位。而串行接口每次只能发送或接收1位数据。总线上的通信对于两种方式都是一样的，并行和串行之间的转换在接口电路中完成。

在并行端口中，设备和计算机之间通过多引脚连接器和有多条电线的电缆连接，一般排列成平板结构。两端的电路相对简单些，因为这里不需要进行并行和串行转换。这种结构只适用于物理上与计算机相近的设备。对于较长的距离，前面提到的时序相位偏移问题就会限制可用的数据传输速率。在一些需要更长电缆的地方，串行方式要方便和有效得多。串行传输形式将在第10章中讲述。

在讲述具体的接口电路之前，首先复习一下接口电路的功能。根据4.1节的内容，I/O接口：

1. 提供至少一个数据字的存储缓冲空间（或一个字节，在面向字节的设备中）。
2. 包含状态标志，处理器通过访问该状态标志确定缓冲区是否已满（对输入）或已空（对输出）。
3. 包含地址译码电路以确定何时被处理器寻址。
4. 产生总线控制所需要的定时信号。
5. 执行所有在总线和I/O设备之间传送数据所需的格式转换，如串行端口下的并/串转换。

### 4.6.1 并行端口

现在我们用一个实际例子给大家解释一下接口设计的主要内容。首先，描述一下分别用于8位输入端口和8位输出端口的电路。然后，将这两种电路结合起来介绍如何设计一个通用8位并行端口的接口。假定这个接口是连接到一个32位处理器上的，该处理器使用存储器映射I/O寻址以及图4-26与图4-27所描述的异步总线协议。最后介绍如何修改该设计使它适应图4-25中的总线协议。

248

图4-28显示了将一个键盘连接到处理器所需要的硬部件。一般的键盘由机械开关组成，这些开关通常是断开的。当一个键被按下时，它的开关闭合形成一条电信号通路。该信号可以被编码器电路检测到然后产生相应字符的ASCII代码。这种按钮开关的一个难点是当一个键被按下后的

接触反弹。即使反弹只持续1~2毫秒，这也足以使计算机将一次按键行为监测为几次不同的电气事件。这个单次按键行为将被错误地认为是已经快速地按下和松开了多次，所以必须消除反弹的副作用。这里有两种方法可以解决该问题：使用一个简单的消除反弹电路，或是使用软件。当采用软件方法来消除反弹时，I/O程序从键盘读入一个字符后等待足够长的时间来确保反弹已经结束。图4-28所示的是硬件方法，消除反弹的电路包括在编码器模块中。

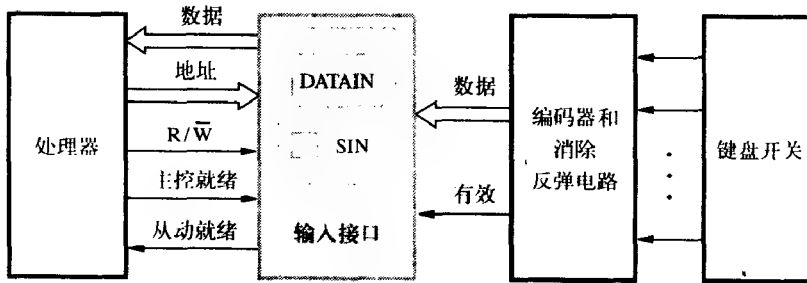


图4-28 键盘与处理器的连接

编码器的输出由表示编码字符的位序和一个有效（Valid）控制信号组成，该控制信号表示一个键正被按下。将这些信息发送到接口电路上，接口电路中有一个数据寄存器DATAIN和一个状态标志SIN。当一个键被按下时，有效信号从0变为1，并将ASCII码装入DATAIN，同时将SIN置1。处理器读取DATAIN寄存器的内容后状态标志SIN清0。这个接口电路与一条异步总线相连，该总线使用图4-26的握手信号“主控就绪”和“从动就绪”控制传输，此外还有第三根控制线R/  $\bar{W}$ 用来区分读写传输。

图4-29是一个输入接口电路。DATAIN寄存器的输出线通过三态驱动器连接到总线的数据线上，当处理器发出包含有该寄存器地址的读指令时，三态驱动器被打开。SIN信号由状态标志电路产生，也通过三态驱动器发送到总线上。SIN与位D0相连，也就是说它将作为状态寄存器的第0位，状态寄存器的其他位不包含有效的信息。当地址的高31位与分配给这个接口的任一地址相符时，由地址译码器选定该输入接口。在主控就绪信号有效时，地址A0位决定是读取状态寄存器还是数据寄存器。在读取状态信号或读取数据信号等于1时，握手控制通过激活从动就绪信号来完成。

249

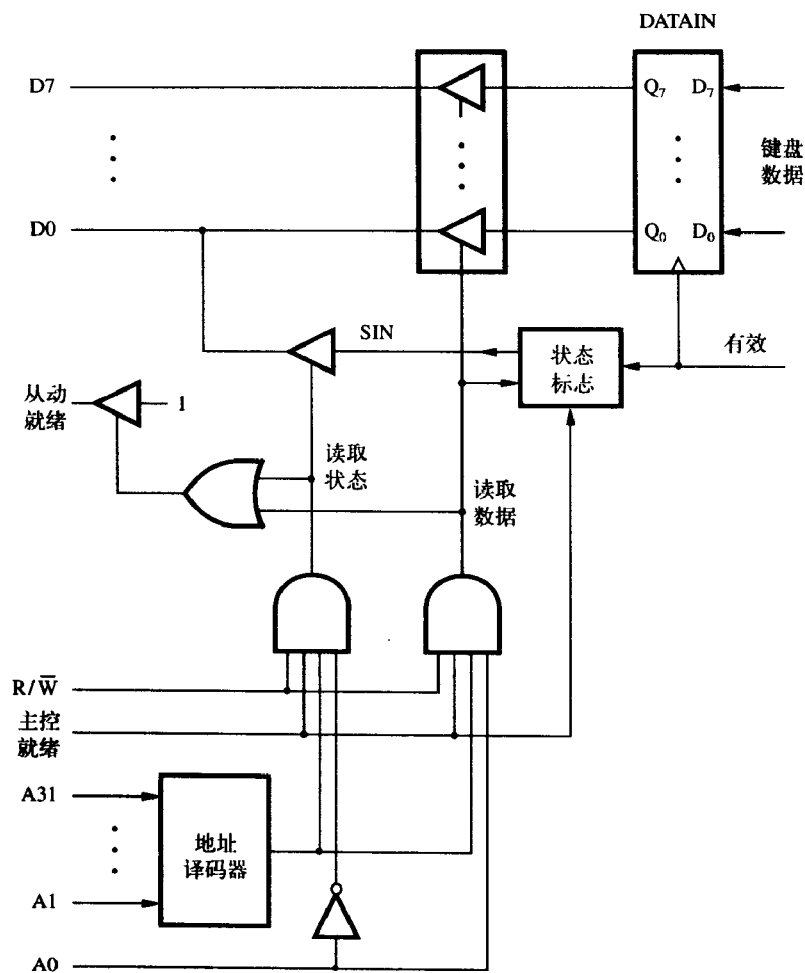
图4-30是状态标志电路的一种可行实现方案。图中边缘触发的D触发器将被有效信号线上的上升沿置1，NOR锁存器的状态也将跟着改变，从而SIN被置1。在SIN被处理器读取的时候，锁存器的状态不允许改变。因此，该电路就可以确保只有当主控就绪信号等于0时才可以对SIN进行设置。当读取数据信号被置1，读取DATAIN寄存器时，触发器和锁存器都被重置成0。

250

现在让我们再来看一个可以用来将输出设备（比如打印机）连接到处理器的输出接口，如图4-31所示。打印机在握手信号“有效”和“闲置”的控制下工作，控制方式与总线上使用主控就绪和从动就绪信号的握手方式相似。当准备好接收一个字符时，打印机启动它的闲置信号，然后接口电路将一个新字符放置到数据线上并激活有效信号。打印机进行响应，打印一个新字符并取消闲置信号，接口电路也紧接着撤销有效信号。

接口中有一个数据寄存器DATAOUT和一个状态标志SOUT。SOUT标志在打印机准备好接收另一个字符时被置1，当处理器将一个新字符装入到DATAOUT后被清0。图4-32显示了这种接口的一种实现。它的操作与图4-29中的输入接口相似。惟一显著的区别是握手控制电路，具体的设计作为一个练习留给读者。

251



· 图4-29 输入接口电路

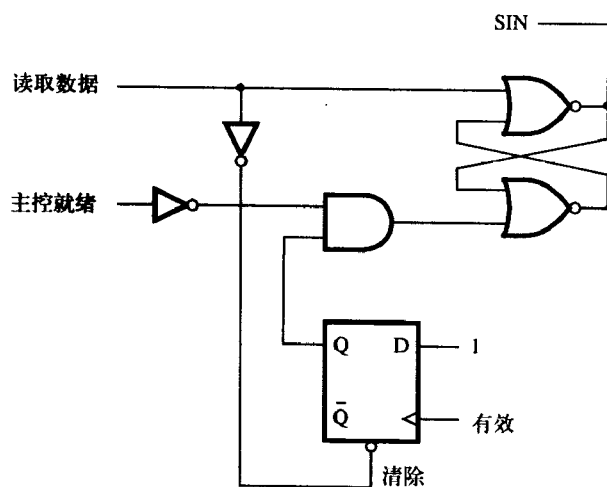


图4-30 图4-29中状态标志模块的电路

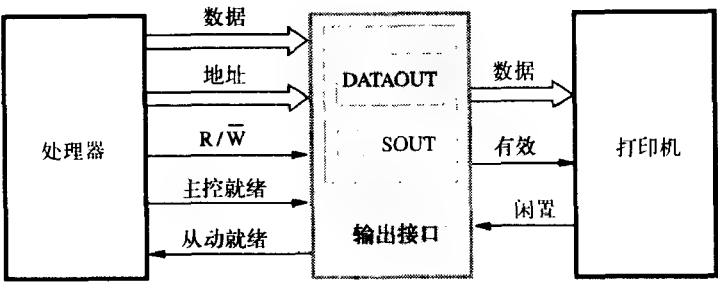


图4-31 打印机与处理器的连接

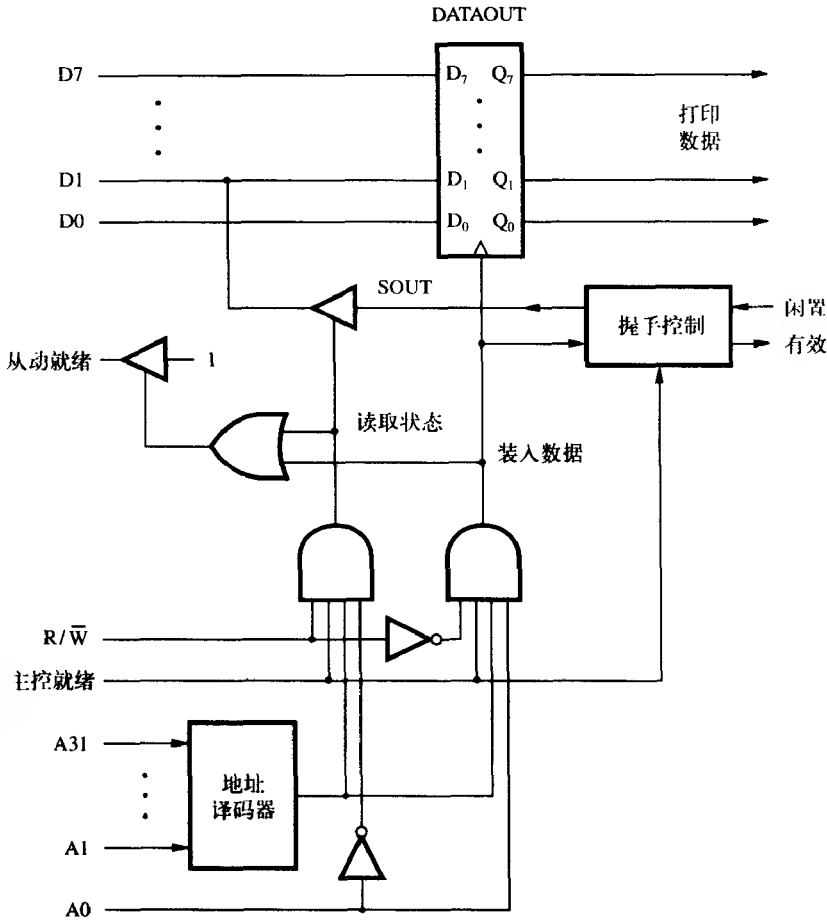
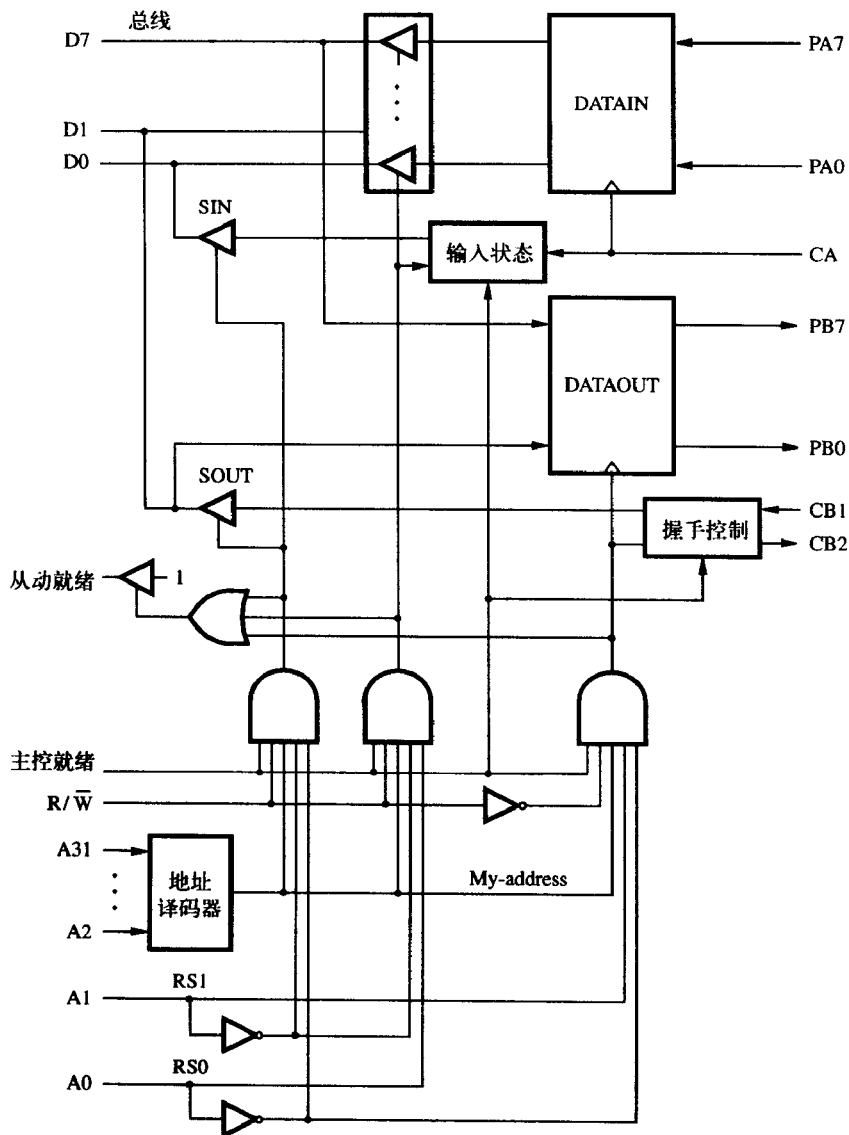


图4-32 输出接口电路

上面讲述的输入和输出接口可以结合成为一个接口，如图4-33所示。此时，整个接口由地址信号的高30位来选定。地址位A1和A0用于选择接口中三个可编址单元中的一个，即两个数据寄存器和一个状态寄存器。状态寄存器包括标志SIN和SOUT，分别为0位和1位。因为在I/O接口中，这些位置常常为寄存器，所以我们用符号RS1和RS0来标志两个输入，它们决定寄存器的选取。

图4-33中的电路使用独立的输入和输出数据线来连接I/O设备。如果连接I/O设备的数据线是双向的，那么得到的并行端口将会更加复杂。图4-34是一个通用并行接口电路，它可以按多种方

法进行配置。数据线P7到P0既可以用作输入也可以用作输出。为了增加灵活性，电路允许在程序控制下将一些线路用于输出，一些线路用于输入。DATAOUT寄存器通过数据方向寄存器DDR控制的三态驱动器连接到数据线上。处理器可以将任意的8位序列写入DDR。对某一特定的位，如果DDR中的值为1，相应的数据线路就作为输出线，否则就作为输入线。



253

图4-33 将输入输出结合在一起的接口电路

C1和C2用来控制接口电路和它服务的I/O设备之间的相互作用。这两根线也是可编程的。C2是双向的，可以提供几种不同的发信号模式，包括握手。图中并没有显示所有的内部细节，但可以看出这些部分是如何与图4-33对应的。“就绪”和“接受”线是处理器总线端的握手控制线，因此它们和主控就绪和从动就绪信号相连。输入信号My-address应该连接到地址译码器的输出上，地址译码器将识别出分配给接口的地址。共有3条寄存器选择线，可以允许接口中最多8个寄存器，

如输入输出数据、数据传输方向和各种操作模式下的控制和状态寄存器。还提供有中断请求输出线  $\overline{\text{INTR}}$ ，它连接到计算机总线的中断请求线上。

实际中我们常常用到具有图4-34所显示特性的并行接口电路。第9章中将讲述一个在嵌入式系统中使用这种接口电路的例子。在该系统中连接I/O设备的端口不止一个，可以有2到更多个。

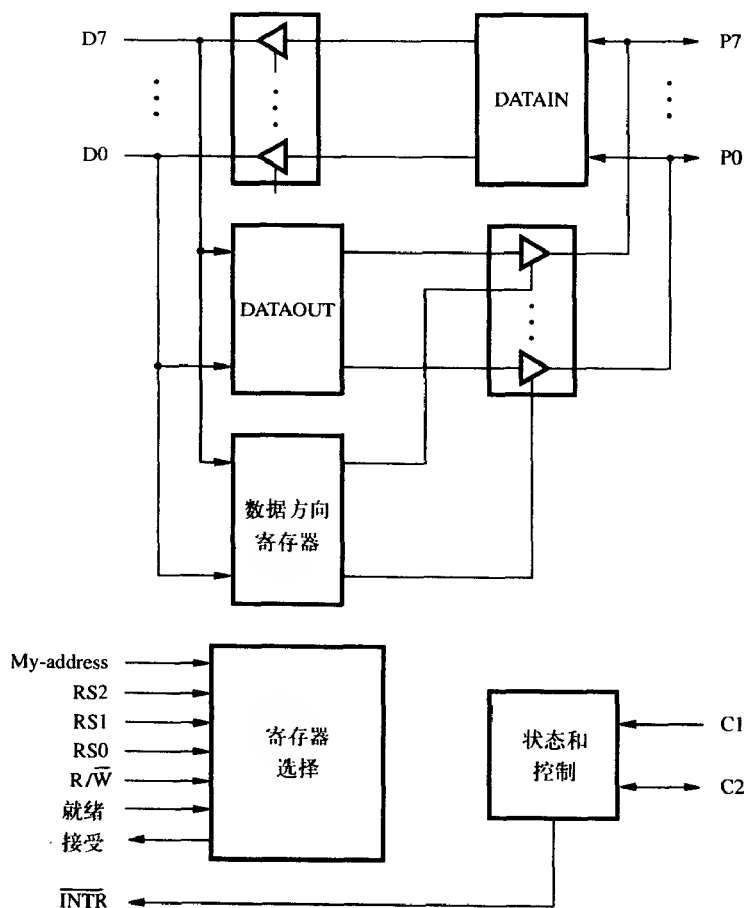


图4-34 通用8位并行接口

254

现在来分析一下图4-28到图4-34中的接口电路是如何被修改来符合图4-25中的总线协议的。图4-35是图4-32中接口的一种修改电路。前面我们已经介绍了产生装入数据和读取状态信号的时序逻辑模块。在图的底部给出了这个模块的状态图。起初，接口处于闲置状态。当地址译码器的输出My-address显示该接口被寻址后，接口改变状态进行响应。于是，接口启动Go信号，并根据地址位A0和 $\text{R}/\overline{\text{W}}$ 线的状态，启动装入数据或读取状态信号。

图4-36显示了一个输出操作的时序图。处理器在时钟周期1同时发送数据和地址。时序逻辑在时钟周期2开始时将信号Go置1，该信号的上升沿将输出数据装入到寄存器DATAOUT。读取状态寄存器的输入操作也遵照相似的时序。因为请求的数据在寄存器中已经可用，并能立即进行传送操作，所以时序逻辑模块直接从闲置状态转换到响应状态。结果，这个传输操作要比图4-25中的短一个时钟周期。在有些情况下，在数据可用前可能需要一些时间，此时接口应该首先进入等待状态，等到数据准备就绪后才能转换到响应状态。



式通信。并行与串行之间的转换由移位寄存器完成，移位寄存器有并行存取的能力。图4-37是一个典型的串行接口模块图。它包括常见的DATAIN和DATAOUT寄存器。输入移位寄存器接收来自I/O设备的串行位输入。当8位数据都接收完后，移位寄存器的内容并行装入到DATAIN寄存器中。相似地，DATAOUT寄存器中的输出数据也被装入输出移位寄存器，在输出移位寄存器中数据被逐位移出并发送到I/O设备上。

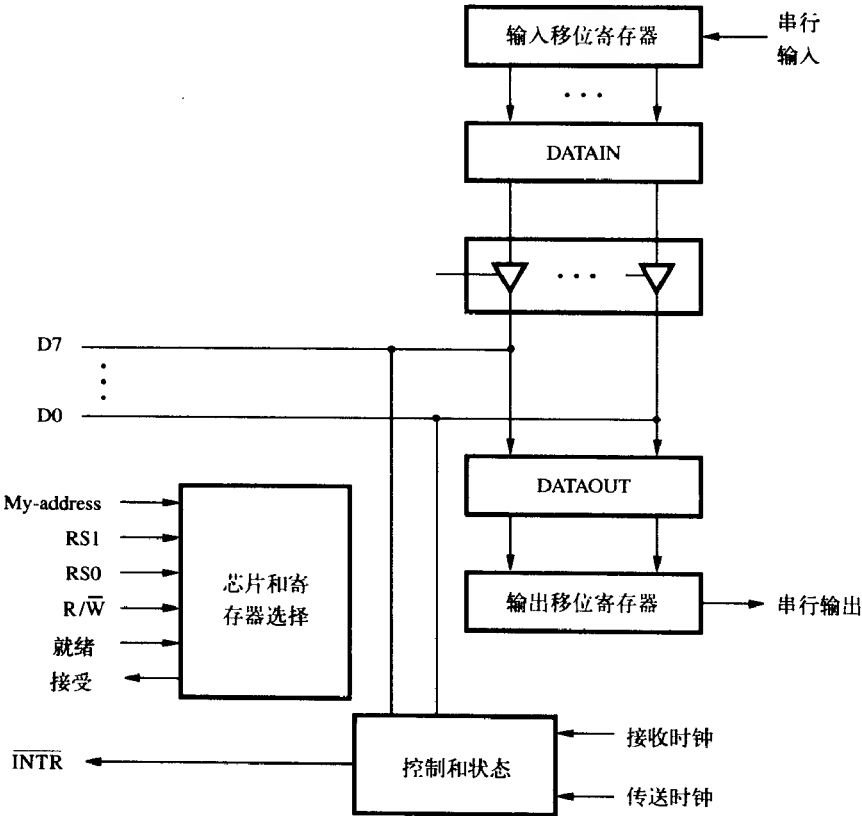


图4-37 串行接口

这种接口中涉及的总线部分与前面讲述的并行接口相同。状态标志SIN和SOUT的功能也与并行接口中的相似。当新数据被装入到DATAIN后SIN标志置1，处理器读取了DATAIN中的内容后清0。一旦数据从输入移位寄存器传送到DATAIN寄存器，移位寄存器就可以开始接收来自I/O设备的下一个8位字符。SOUT标志指出输出缓冲区是否可用，处理器将新数据写入到DATAOUT寄存器后被置0，数据从DATAOUT寄存器传送到输出移位寄存器后被置1。

257

输入输出通路上使用的两个缓冲区非常重要。在一些简单的接口中可能将DATAIN和DATAOUT移到了移位寄存器的单元中，而删除了图4-37中的移位寄存器。但是，这将会给I/O设备的操作带来不便，在接收完来自串行线上的一个字符后，设备只有等到处理器读取了DATAIN的内容后才能开始接收下一个字符。因此，两个字符之间就需要一个间隔让处理器来读取输入数据。如果有两个缓冲区的话，第二个字符的传送在第一个字符从移位寄存器装入到DATAIN寄存器后就可以开始。因此，如果假设处理器在第二个字符的串行传输完成前读取了DATAIN寄存器的内容，接口就可以接收连续的串行数据流。在接口的输出通路中也存在类似的情况。

258

因为串行传输只需要很少的电路，所以在连接物理上与计算机相距非常远的设备时很方便。



传输的速度，通常用位速率来表示，它取决于连接设备本身的特性。为了适应各种各样的设备，串行接口必须能够使用各种不同的时钟速度。图4-37中的电路允许对输入和输出操作分别使用独立的时钟以提高灵活性。

因为串行接口在连接I/O设备中起着至关重要的作用，所以已经定义了多种广泛使用的标准。具有图4-37特性的标准接口电路称为通用异步收发器（UART），它主要用于低速串行设备。数据传送采用的是异步起止方式，起止方式将在第10章讲述。为了便于连接到通信链路上，开发出了流行的RS-232-C标准接口，这些也将在第10章详细描述。

## 4.7 标准I/O接口

前面几节中我们曾指出计算机总线有多种可供选择的设计。这种多样性意味着配备有这种接口电路的I/O设备适用于一台计算机不一定就适用于另一台计算机。因此，每一种I/O设备与计算机的连接可能都需要设计一种不同的接口，这样就会产生很多不同的接口。最符合实际的解决办法就是开发出标准的接口信号和协议。

这里首先来了解一下计算机系统是如何被组装起来的。通常个人计算机都包括有一个称为主板的印刷电路板。在这块板上安装有CPU芯片、主存以及一些I/O接口。它还包含一些连接器，可以插入额外的接口。

处理器总线是由处理器芯片自己定义的信号。那些需要连接到处理器的高速设备，如主存，可以直接连接到这些总线上。由于电气方面的原因，只有很少的设备可以用这种方式连接。主板中通常提供另外的总线来支持更多的设备。这两种总线通过一个电路相互连接起来，这个电路我们称作为桥，它将一种总线的信号和协议转换成另一种总线的信号和协议。连接到扩展总线上的设备，在处理器看来就好像是连接到它自己的总线上一样。惟一的不同之处是，在处理器与那些设备之间的数据传送中，桥电路会产生一个小小的延迟。

总线结构与处理器的体系结构密切相关，而且还依赖于处理器芯片的电气特性，如时钟速度，所以不可能为处理器总线定义一个统一标准。但扩展总线就没有这些限制了，因此可以使用标准的信号方式。已经有许多标准被开发出来了。其中有些标准是默认的，当特定的设计在商业上取得成功时，这些设计就成为了事实上的标准。例如，IBM为它的个人计算机PC AT开发的总线ISA（工业标准体系结构），由于这种计算机的流行导致其他制造商为他们的I/O设备生产出与ISA兼容的接口，这样ISA就成为了事实上的标准。

还有一些标准是行业共同协作开发的，由于在兼容产品上的共同切身利益，即使那些互相竞争的公司也参与了标准的开发。IEEE（电气和电子工程师协会）、ANSI（美国国家标准学会）等组织及一些国际组织如ISO（国际标准化组织）已经承认了这些标准，并给它们授予官方的地位。

在这一节中，我们讲述三种广泛使用的标准：PCI（外围部件互连）、SCSI（小型计算机系统接口）和USB（通用串行总线）。图4-38显示了这些标准在一般计算机系统中的使用方法。PCI标准定义了主板上的一种扩展总线。SCSI和USB标准用来连接机箱内外的附加设备。SCSI总线是高速并行总线，用于像硬盘和视频显示器之类的设备。USB总线使用串行传输，适合键盘、游戏控制器和Internet连接等设备的需求。图中显示了一个可以与早期的ISA标准连接相兼容的设备接口电路，比如流行的IDE（集成设备电子电路）磁盘。还显示了一个与Ethernet的连接。Ethernet是一种广泛使用的局域网，为建筑物或大学校园里的计算机之间提供高速互连。

一台特定的计算机使用的标准可能不止一个。典型的Pentium计算机有PCI和ISA两种总线，以提供大范围的设备给用户选择。

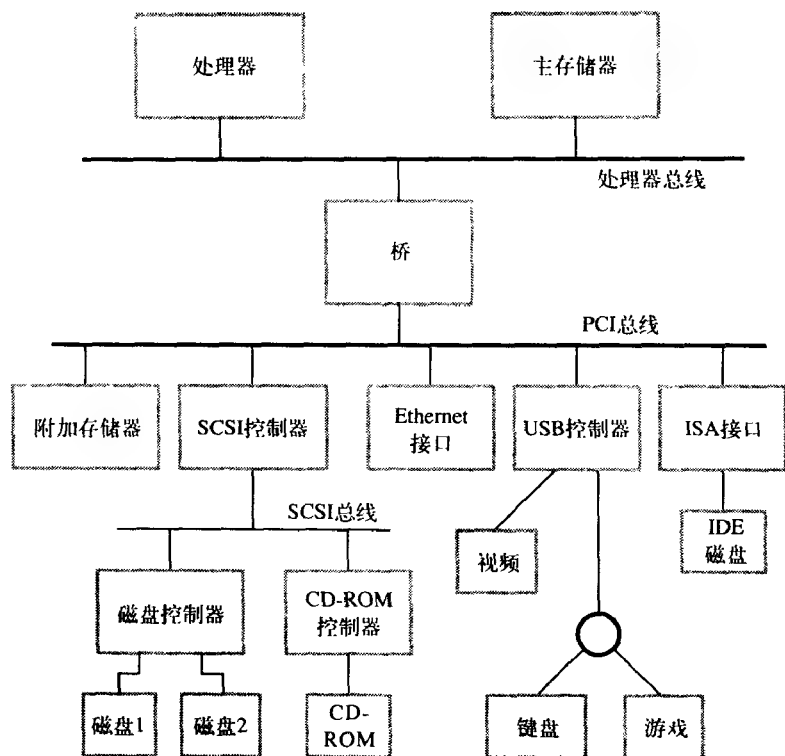


图4-38 一个使用不同接口标准的计算机系统的例子

260

#### 4.7.1 外围部件互连（PCI）总线

PCI总线<sup>[1]</sup>是那些由于标准化的要求而产生的系统总线中的一个很好的例子。它支持处理器的功能，但又以一种标准的形式独立于任何特定的处理器。连接到PCI总线的设备在处理器看来就好像它们是直接连接到处理器总线上的一样。它们在处理器的存储器地址空间内分配地址。

在PCI标准之前还有一系列主要用于IBM PC上的总线标准。早期的PC使用8位XT总线，它的信号和Intel 80x86处理器的信号十分相似。此后，16位的总线开始用于PC AT 计算机，即ISA总线，它的32位扩展版本称为EISA总线。其他在20世纪80年代开发的具有相似功能的总线还有在IBM PC中使用的Microchannel和在Macintosh计算机上使用的NuBus。

PCI是一种廉价且真正独立于处理器的总线。它的设计预见了对支持高速磁盘、图形和视频设备而引起的对总线带宽需求的快速增长，也预见了对多处理器系统专业化需求的快速增长。PCI总线自从1992首次被推出，近十年来作为一个工业标准一直非常流行。

PCI总线首创的一个重要特性是I/O设备的即插即用。要连接一台新的设备，用户只需将设备接口板连接到总线即可，其余的操作由软件来完成。在讲述PCI总线如何工作时我们将讨论该特性。

##### 数据传送

在当代计算机中，大部分的存储器传输中都包含一组数据而不只是一个字。原因是现代计算机都有缓冲存储器（见图1-6）。数据在缓冲存储器和主存之间以几个字的块形式进行传输，这将在第5章我们将进行解释。传输的字存储在连续的存储器单元中。当处理器（实际上是缓存控制器）

指定一个地址请求一次对主存的读操作时,存储器就从该地址开始发送一个数据字序列进行响应。类似地,在写操作中,处理器首先发送一个存储器地址,然后将其后面一个数据字序列写入到从该地址开始的连续存储器区域中。PCI主要是为了支持这种操作模式而设计的。只包括单个字的读写操作可简单地认为是长度为1的块。

**261** PCI总线支持三个独立的地址空间:存储器、I/O和配置。前两个从名称上我们就可以知道它们的意思。I/O地址空间主要用于那些有独立的I/O地址空间的处理器,如Pentium。然而,就如第3章提到的,即使有独立的I/O地址空间可用,系统设计者也可以选用存储器映射I/O寻址。实际上,PCI总线就是建议使用这种方法以获得更广泛的兼容性。配置空间是用来支持PCI即插即用功能的,后面将简要说明。伴随地址的一个4位指令将指明在特定数据传输操作中使用的是哪一个地址空间。

在图4-38中计算机主存直接连接到处理器总线上。图4-39中显示了另一种常与PCI总线一起使用的结构。PCI桥为主存提供了一个独立的物理连接。由于电气原因,总线可能被进一步分成几个部分并通过桥连接起来。但是,不管设备连接到总线上的哪一部分,它都可能被映射到处理器的存储器地址空间中。

PCI总线上的信号约定类似于图4-25中使用的方式。在图4-25中,我们假设主控设备将在总线上保持地址信息直到数据传输结束。但这并不是必须的,地址信息只需保持到从动设备被选定就足够了。从动设备可以将地址存储在它的内部缓冲区中。因此,地址在总线上只需保持一个周期即可,这样就可以释放地址线,使它在后续时钟周期中用于发送数据。

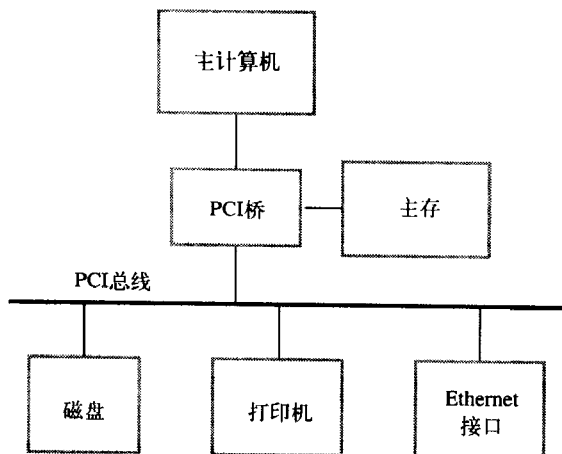


图4-39 PCI总线在计算机系统中的应用

**262** 这样可以节约大量成本,因为总线的线路数量是一个重要的成本因素。PCI总线使用的就是这种方法。

在任何时候都有一台设备是总线主控器。它可以通过发布读写命令启动数据传送。主控设备在PCI术语中称为启动设备,它可以是处理器或DMA控制器。响应读写命令的被寻址设备称为目标设备。

为理解PCI总线的操作和它的各种特性,下面我们将分析一个典型的总线事务。表4-3中列出了用于传输数据的主要信号。以#结尾命名的信号说明低电平有效。PCI协议与图4-25的主要区别是除了目标就绪信号外,PCI还使用了起始就绪信号IRDY#。IRDY#信号主要用于支持块传送。

让我们来看一个处理器从存储器中读取4个32位字的总线事务。在这个事务中,启动设备是处理器,目标设备是存储器。总线上一次完整的传送操作,包括一个地址和一块数据,称为事务。事务内单个字的传送称为相位。图4-40显示了总线上的事件顺序。时钟信号提供时序参考来协调事务中的不同相位。所有信号的跳变由时钟的上升沿触发。与图4-25一样,通过显示信号在这个时钟周期中稍后才发生变化来表示它们遇到的延迟。

表4-3 PCI总线上的数据传送信号

名 称	功 能
CLK	33MHz或66MHz的时钟
FRAME#	由启动设备发送、表示事务的持续时间
AD	32位地址/数据线，可以增加至64位
C/BE#	4位指令/字节允许线（64位总线时为8位）
IRDY#,TRDY#	起始就绪信号和目标就绪信号
DEVSEL#	来自设备的响应，表示它已经识别了地址并准备好了数据传送事务
IDSEL#	初始化设备选择

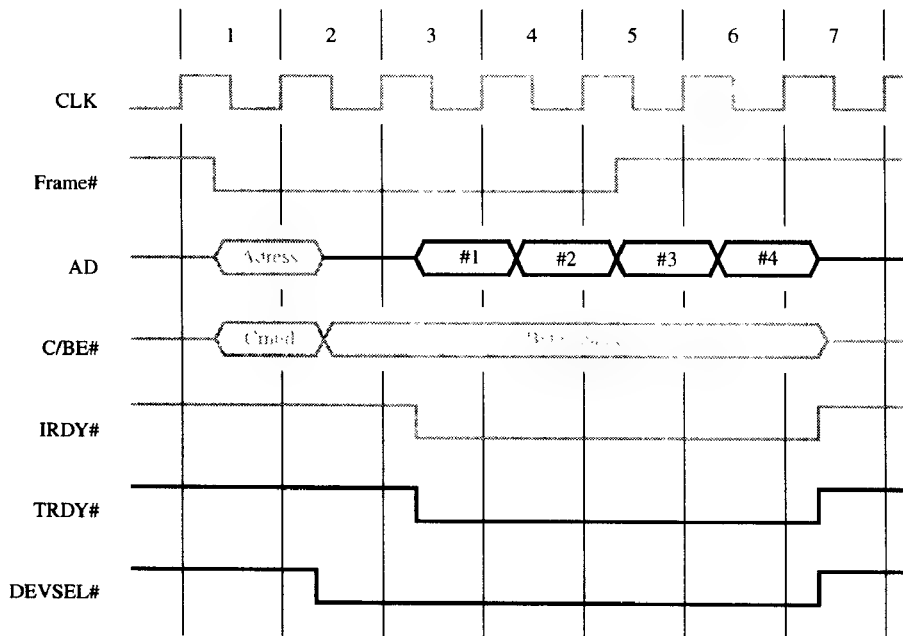


图4-40 PCI总线上的读操作

在时钟周期1，处理器启动信号FRAME#表示事务开始。同时，它将地址发送到AD线上，将指令发送到C/BE#线。在本例中，指令指明了这是一次读操作请求和使用的存储器地址空间。

时钟周期2用来转换AD线。处理器撤销地址信号并将它的驱动器从AD线上断开。被选定的目标设备将它的驱动器连接到AD线上，并在时钟周期3将被请求的数据取出来放到总线上。它还启动DEVSEL#信号并保持到事务结束。

263

C/BE#线在时钟周期1用来发送总线指令，在其余周期中用作其他目的。四根线中的每一根都与AD线上的一个字节相关联。启动设备设置C/BE#线中的一根或几根来表示哪几根字节线被用来传送数据。假设目标设备每次可以传送32位，那么四根C/BE#线都有效。

在时钟周期3，启动设备发送起始就绪信号IRDY#，表示它已经准备好接收数据了。如果目标设备此时也已经准备好发送数据，它就启动目标就绪信号TRDY#，并发送1个字的数据。启动设备在这个时钟周期结束时将数据装入到输入缓冲区。目标设备在时钟周期4到6发送其余3个字的数据。

启动设备使用FRAME#信号表示脉冲的持续时间。它在传送倒数第二个字时取消这个信号。

因为本事务要求读取4个字的数据，所以启动设备在时钟周期5，也就是在它接收第3个字的周期取消FRAME#信号。在时钟周期6发送完第4个字后，目标设备断开它的驱动器并在时钟周期7开始时取消DEVSEL#信号。

图4-41给出了一个更一般的输入事务的例子。它显示了启动设备和目标设备是怎样分别使用IRDY#和TRDY#信号来表示事务中间的停顿的。这个读操作的开始与图4-40一样，前两个字被传送。目标设备在周期5发送第3个字。然而，我们假设此时启动设备不能接收第3个字，所以它就取消了IRDY#信号。相应地，目标设备就在AD线上保持第3个字直到IRDY#再次有效。在周期6，启动设备启动IRDY#信号并在这个周末将数据装入到它的输入缓冲区中。在这里，我们假设目标设备没有立即准备好传送第4个字，因此它在第7个周期开始时取消TRDY#信号。在周期8，它发送第4个字并启动TRDY#。因为FRAME#在传输第3个字时就已经取消了，所以在传输完第4个字符后事务就结束了。

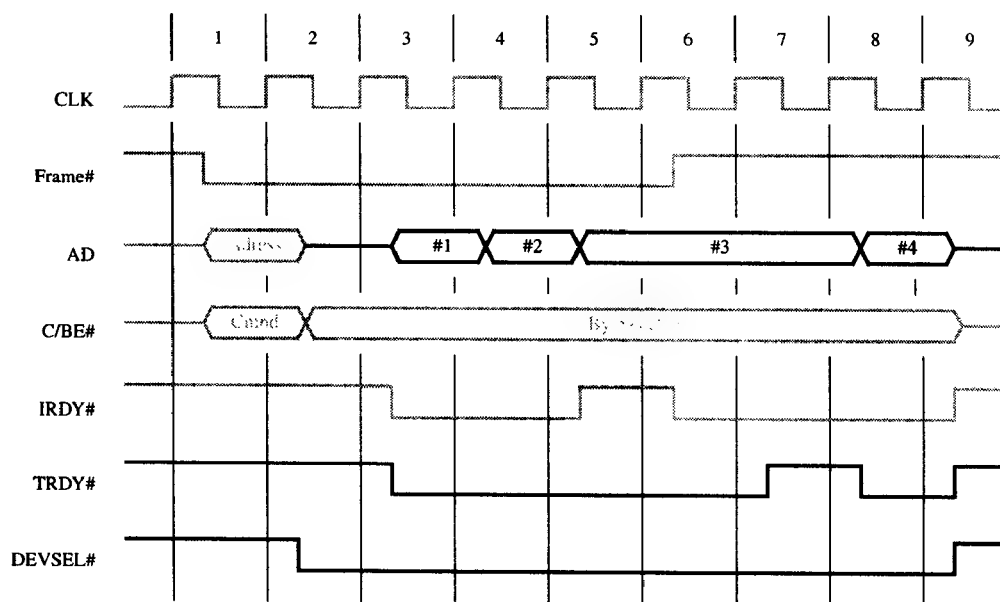


图4-41 显示IRDY#/TRDY#信号作用的读操作

### 设备配置

将一台I/O设备连接到计算机上时，需要对与它进行通信的设备和软件进行配置操作。例如，典型的ISA总线的设备接口卡，就有大量的跳线或开关需要用户来配置。一旦设备连接到了计算机上，软件就需要知道设备的地址，可能还需要知道一些相关的设备特性，如传输连接的速度、是否使用了奇偶校验位等。

PCI为每一个I/O设备接口分配一个较小的配置ROM存储器来保存有关设备的信息，这样简化了配置过程。所有设备的配置ROM在配置地址空间内都可以被访问。PCI的初始化软件在系统启动或复位时读取这些ROM的内容，根据这些内容确定该设备是打印机、键盘、Ethernet接口或磁盘控制器，还可以得知各种设备选项和特性。

设备的地址是在初始化过程中分配的。也就是说在总线配置操作期间，不能根据它们的地址访问设备，因为还没有给它们分配地址。因此，配置地址空间使用了另一种不同的机制。每台设

备都有一个输入信号称为初始化设备选择IDSEL#。在配置操作期间,正是这个信号,而不是施加在设备AD输入上的地址信号,使得这个设备被选定。设备的连接器插在主板的上面,一般都将每台设备的IDSEL#引脚连接到高位的21根地址线上,AD11到AD31上。因此,我们可以通过发布一个配置命令和一个地址来选定一台设备进行配置操作,在发布的地址中相应的AD线被置1,其余的20根线被置0。低位地址线AD10到AD00用来说明操作的类型和访问设备配置ROM的内容。这种结构将设备数量限制在21台以内。

配置软件扫描配置地址空间内的所有21个位置确定安装了哪些设备。每台设备可以在I/O或存储器空间申请一个地址。将这个地址写入到适当的设备寄存器后,设备就和相应的地址相关联了。配置软件也设置一些参数如设备中断优先级等。PCI总线有四根中断请求线。通过写入设备配置寄存器,配置软件指示设备使用哪一根线请求中断。如果设备需要初始化,初始化代码就存储在设备接口的ROM中(这个ROM与配置过程中使用的不同)。PCI软件读取该代码并执行它来完成所要求的初始化操作。

这个过程将用户从必须进行相关配置的操作中解放出来。使用者只需插上接口板并打开电源即可,其余的工作全部由软件来完成。

PCI总线在PC领域获得了极大的流行。它也用在许多其他的计算机中,比如SUN,这些主要得益于它可以为大范围的I/O设备提供PCI接口。在一些处理器中,如Compaq Alpha, PCI处理器桥电路被集成在处理器芯片中,这就更简化了系统的设计和封装。

#### 电气特性

规定PCI总线在5V或3.3V电源下工作。主板可以在任何信号系统下工作。扩展板上连接器的设计可以确保只有在兼容的主板上才可以将它们插进去。

### 4.7.2 SCSI总线

缩写SCSI代表小型计算机系统接口。它是由美国国家标准组织(ANSI)根据设计X3.131<sup>[2]</sup>定义的一种总线标准。在最初规定中,磁盘等设备通过50根线的电缆连接到计算机中,电缆最长可达25米,数据传输速率最高可达5MB/s。

266

SCSI总线标准经历了多次修改,它的数据传输能力增长得非常快,几乎每两年就翻一倍。SCSI-2和SCSI-3已经被定义,每一种都有好几个选项。SCSI可能只有8根数据线,称为窄总线,每次只能传输一个字节的数据。此外,有16根数据线的宽总线每次能传输16位。使用的电信号方式也有好几种选择。SCSI总线可以使用单端传输(SE),此时每一个信号使用一根线,对于所有信号使用一个共用的地进行返回。在另一种选择中,使用了差分信号,对每一个信号都提供有单独的返回线。这种方式有两种电平可以使用。早期的标准使用5V(TTL电平)称为高电平差分(HVD)。最近,又推出了3.3V的标准称为低电平差分(LVD)。

由于有这些不同的选项,所以SCSI连接器可能有50、68和80个引脚。目前流行的商用设备的最大传输速率在5MB/s到160MB/s之间。SCSI标准的最新版本可支持高达320MB/s的传输速率,640MB/s的版本不久也有望出现。具体总线上的最大传输速率通常是电缆长度和连接设备数量的函数,长度越短,设备越少速率就越高。为了获得最高数据传输速率,总线长度一般限制在1.6m(SE方式)和12m(LVD方式)以内。但是,制造商常常还提供专门的总线扩展器来连接远距离的设备。这种总线可容纳的最大设备数量,窄总线为8台,宽总线为16台。

连接到SCSI总线的设备不属于处理器的地址空间,这与连接到处理器总线的设备是一样的。

SCSI总线通过SCSI控制器连接到处理器总线上,如图4-38所示。这个控制器使用DMA方式在主存和设备之间传送数据包。数据包可能是一个数据块,也可能是处理器发送给设备的命令或设备的状态信息。

为了说明SCSI总线的操作,让我们来看一下它是如何与磁盘驱动器一起工作的。与磁盘驱动器通信与跟主存通信在本质上是完全不同的。如第5章所述,数据是存储在磁盘的扇区中的,每个扇区包括几百个字节。这些数据没有必要存储在相邻的扇区中,因为磁盘中一些扇区可能已经存储了数据,还有一些可能有缺陷必须忽略。所以,一次读或写请求访问的可能是几个不相邻的磁盘扇区。由于磁盘机械移动的限制,在到达第一个需要传送数据的扇区前可能会有几毫秒的较长延迟,然后一段数据被高速传送。接着又会发生另一个延迟,传送另一段数据。一次读或写请求可能包括好几段这样的数据。SCSI协议有助于这种操作模式的实现。

267 连接到SCSI总线的控制器是如下两种类型之一——启动控制器或目标控制器。启动控制器可以选择一台特定的目标控制器并发送指令指明将要执行的操作。很明显,处理器这一侧的控制器,如图4-38中的SCSI控制器,必须可以作为启动控制器进行工作。磁盘控制器是作为目标控制器进行工作的,它执行来自启动控制器的命令。启动控制器将与选定的目标控制器建立一个逻辑连接。连接建立后,可以根据传送命令和数据块的需要将其中断和恢复。当一个连接被中断后,其他设备可以使用总线传送数据。重叠数据传送请求的能力是SCSI总线的主要特性之一,它使SCSI总线具有较高的性能。

SCSI总线上的数据传送总是由目标控制器控制的。为了发送一个命令给目标设备,启动控制器请求总线控制权,仲裁获胜后启动控制器选择要进行通信的控制器并将总线控制权转交给它,然后控制器启动一次数据传送操作接收来自启动设备的命令。

下面分析一个完整的磁盘读操作例子。这里虽然我们认为启动控制器是操作的发起者,但应该清楚的是,只有接收到处理器相应的命令后,启动控制器才执行这些操作。假设处理器要从磁盘驱动器读取一块数据时,这些数据存储在两个不相邻的磁盘扇区中。处理器发送一个命令给SCSI控制器,将发生如下事件序列:

1. SCSI控制器作为启动控制器竞争总线控制权。
2. 当启动控制器在仲裁过程中获胜后,它选择目标控制器并将总线控制权转交给它。
3. 目标设备启动一次输出操作(从启动设备到目标设备);启动设备响应,发送一个命令来说明要求的读操作。
4. 目标设备认识到它首先需要执行磁盘查找操作,于是就发送一条消息给启动设备表明它要暂时中断连接,然后释放总线。
5. 目标控制器发送一个命令给磁盘驱动器移动磁头到读操作中的第一个扇区。然后,读取存储在扇区里的数据并保存到数据缓冲区中。当准备好开始传送数据给启动设备时,目标设备请求总线控制权。仲裁获胜后,它重新选启动控制器,从而恢复被中断的连接。
6. 目标设备传送数据缓冲区的内容给启动设备,然后再次中断连接。数据可以8位或16位的并行传送,这取决于总线宽度。
7. 目标控制器发送一个命令给磁盘驱动器执行另一次查找操作。然后,和上次一样将第二个磁盘扇区的内容传送给启动设备。这次传送结束后,两个控制器之间的逻辑连接终止。
8. 当启动设备控制器接收到数据时,它使用DMA方式将数据存储到主存中。
9. SCSI控制器发送一个中断给处理器通知它请求的操作已经完成。

这一过程表明SCSI总线上交换的消息要比处理器总线上交换的消息更高级一些。这里的“更高级”是指这些消息涉及的操作需要好几步才能完成，这取决于具体的设备。处理器和SCSI控制器都不需要了解数据传送过程中特定设备的具体操作。在上面的例子中处理器不需要参与磁盘查找操作。

268

SCSI总线标准定义了大量的控制消息，这些消息可以在控制器之间交换来操作不同类型的I/O设备。还定义了用来处理在设备操作和数据传送期间发生各种错误和失败情况的消息。

### 总线信号

现在我们从硬件方面来描述一下SCSI总线的操作。表4-4总结了总线信号。为简单起见，只显示了窄总线（8根数据线）的信号。这里所有信号名称前都有一个减号，这表示在低电平状态时信号有效或数据线等于1。SCSI总线没有地址线。在选择、重选及总线仲裁期间，SCSI使用数据线来识别总线控制器。对于窄总线，最多可以有8个控制器，编号0到7，每一个控制器与同号的数据线相联系。宽总线最多可容纳16个控制器。控制器通过激活相应的数据线将它自己的或其他控制器的地址放到总线上。因此，在总线上可能同时有多于一个的地址，就像下面我们将要讲述的仲裁过程那样。一旦在两个控制器之间建立了连接，就不需要进一步寻址了，数据线就可以用来传输数据了。

表4-4 SCSI总线的信号

类 别	名 称	功 能
数据	-DB(0)到-DB(7)	数据线：在信息传送阶段携带一个字节的信息，在仲裁、选择和重选阶段识别设备数据线的奇偶校验位
状态	-BSY	总线忙：当总线忙时有效
	-SEL	选择：在选择和重选期间有效
信息类型	-C/D	控制/数据：传送控制信息时有效（命令、状态或消息）
	-MSG	消息：指示传送的信息是消息
握手	-REQ	请求：目标设备在请求一个数据传送周期时有效
	-ACK	应答：启动设备完成一次数据传送操作后有效
传输方向	-I/O	输入/输出：有效时表示输入操作（相对于启动设备）
其他	-ATN	注意：当启动设备想要发送一个消息给目标设备时有效
	-RST	复位：所有设备的控制器从总线上断开并恢复为初始状态

269

SCSI总线操作过程中包括的主要阶段有仲裁、选择、信息传送和重选。下面我们将逐一介绍这些阶段。

### 仲裁

当-BSY信号无效时（高电平）总线空闲。此时，任何控制器都可以请求使用总线。但因为可能会有两个或三个设备同时发出请求，所以必须要有仲裁机构。控制器请求总线时启动-BSY信号，并启动与它相关联的数据线来识别自己。SCSI总线使用简单的分布式仲裁方式。图4-42对这种方式进行了说明，在该图中2号到6号控制器同时请求使用总线。

总线上的每一个控制器都分配有一个固定的优先级，其中控制器7的优先级最高。当-BSY有效时，所有请求总线的控制器检查数据线来确定是否有优先级更高的设备同时在请求总线。具有最高优先级数据线的控制器将在仲裁过程中获胜。然后所有其他设备从总线上断开并等待-BSY信号再次变成无效。

在图4-42中，假设控制器6是启动设备要求与控制器5建立连接。仲裁获胜后，控制器6进入



选择阶段识别目标设备。

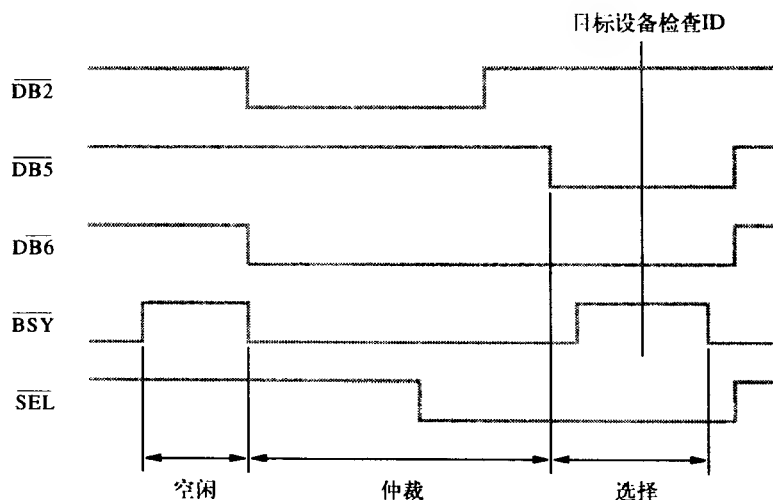


图4-42 SCSI总线上的仲裁和选择过程。设备6在仲裁中获胜且设备2被选中

### 选择

获得总线控制权后，控制器6继续保持 $\overline{\text{BSY}}$ 和 $\overline{\text{DB6}}$ （它的地址）有效，接着它通过启动 $\overline{\text{SEL}}$ 和 $\overline{\text{DB5}}$ 信号表示它要选择控制器5。所有其他参与仲裁过程的控制器，如图中的控制器2，如果还在驱动数据线的话，一旦 $\overline{\text{SEL}}$ 线变成有效必须停止驱动数据线。目标控制器的地址放到总线上后，启动设备释放 $\overline{\text{BSY}}$ 线。

通过启动 $\overline{\text{BSY}}$ 信号被选定的目标控制器进行响应，这个信号通知启动设备它请求的连接已经建立了，这样启动设备就可以从数据线上撤销地址信息。至此选择过程就完成了，此时是目标控制器（控制器5） $\overline{\text{BSY}}$ 信号有效。控制器5控制总线进入信息传送阶段。

### 信息传送

两个控制器之间传送的信息包括：启动设备发送给目标设备的命令、目标设备发送给启动设备的状态信息、发送给或来自I/O设备的数据。使用握手信号按照4.5.2节描述的方式控制信息传送，此时目标设备是总线主控器。 $\overline{\text{REQ}}$ 和 $\overline{\text{ACK}}$ 信号分别取代图4-26和图4-27中的主控就绪和从动就绪信号。目标设备在输入操作（从目标设备到启动设备）期间启动 $\overline{\text{I/O}}$ 信号，还可能启动 $\overline{\text{C/D}}$ 信号表示传送的信息是命令或状态信息而不是数据。

应该指出的是在高速版本的SCSI总线中引入了双边沿时钟或双重传输（DT）技术。在图4-26和图4-27中，每一次数据传送都需要两个握手信号有一次从高到低的跳变，然后接着一次从低到高的跳变。双边沿时钟指的是数据在这些信号的上升沿和下降沿都可以传送，从而使传输速率翻倍。

在传输结束时，目标控制器释放 $\overline{\text{BSY}}$ 信号，从而释放总线给其他设备使用。稍后，当准备好更多要传输的数据时，目标设备要重新建立与启动控制器的连接。这些在重选操作中完成。

### 重选

当目标设备准备好恢复被中断的逻辑连接时，它首先需要获取总线控制权。启动一个仲裁周期，并在仲裁获胜后按与前面描述一样的方式选定启动控制器。但由于目标设备和启动设备的角色已经调换，所以此时启动设备启动 $\overline{\text{BSY}}$ 信号。在数据传送开始前，启动设备必须把控制权转

270

271

交给目标设备。通过使目标控制器在选定启动设备后启动 -BSY信号来完成控制权的转移。同时,启动设备在被选定后需要等待一小段的时间来确定目标设备已经启动 -BSY信号,然后释放 -BSY线。现在两个控制器之间又建立了连接,目标设备控制总线继续传输数据。

上面描述的总线信号方式为两个控制器建立逻辑连接和交换数据提供了必需的机制。这个连接在任何时候都可能被中断和重建。SCSI标准定义了各种类型信息包的结构和内容,控制器通过交换这些信息包来处理不同的情况。启动设备使用这些信息包发送它从处理器接收到的命令给目标设备。目标设备响应,将状态信息发送给启动设备并执行数据传送操作。其中后者是在目标设备的控制下进行的,因为只有目标设备知道什么时候数据准备好了,什么时候中断和重建了连接等。

SCSI总线的其他信息和各种SCSI产品在标准委员会<sup>[2]</sup>的网站上可以查到。

#### 4.7.3 通用串行总线 (USB)

计算机和通信的结合是当代信息技术革命的核心。现代计算机系统一般都包括有多种类型的设备如键盘、麦克风、照相机、扬声器和显示设备。大部分计算机还有接入到Internet的有线或无线连接。在这种环境中,一个关键的要求就是能提供一个简单、廉价的机制将这些设备连接到计算机上,最近在这方面的一个重要突破就是通用串行总线(USB)<sup>[3]</sup>的推出。这是由若干大型计算机和通信公司协同开发出来的一种工业标准,这些公司包括Compaq、Hewlett-Packard、Intel、Lucent、Microsoft、Nortel Networks和Philips。

USB支持两种低速(1.5Mb/s)和全速(12Mb/s)的操作速度。在最近修订的总线规范(USB 2.0)中还引入了第三种操作速度,称为高速(480Mb/s)。USB很快就被市场接受了,并且由于新增的高速功能它已成为大部分计算机设备的选择。

USB的设计达到如下几个主要目标:

- 提供一个简单、廉价、使用方便的互连系统,克服由于计算机只能提供有限数量的I/O端口带来的困难。
- 能够满足I/O设备的多种数据传送特性,包括电话和Internet连接器。
- 采用“即插即用”的操作模式,使用户操作更加方便。

在讲述USB的技术细节之前,我们先详细阐述一下这些设计目标。

##### 端口限制

4.6节中讲述的并行和串行端口为多种低速和中速设备连接到计算机提供了一个通用的连接点。但在实际中,一般的计算机只提供很少的这类端口。为增加新的端口,用户必须打开机箱将一个新的接口卡插入到内部的扩展总线上。而且用户可能还需要知道如何配置设备和相应的软件。USB设计目标之一就是不开机箱就可以随时添加大量设备到计算机上。

##### 设备特性

连接到计算机上的各种设备其功能范围非常广泛。设备的数据传送速度、容量和时序限制相差都非常大。

在键盘上,每按下一个键就会产生一个字节的信

息,而且随时都可能发生。这些信息应该迅速地传送给计算机。因为按键事件不与计算机系统中的任何其他事件同步,所以键盘产生的数据是异步的,并且又受操作者速度的限制,所以产生数据的速率非常低,大概为100 byte/s,少于1000 bit/s。

计算机上有许多简单的附属设备有类似产生数据的特性——低速和异步。计算机鼠标和在视

频游戏中使用的控制器和操纵杆就是很好的例子。

下面看另一种类型的数据源。许多计算机都有外带的或内置的麦克风。这个由麦克风接收到的声音产生了一个模拟电信号，这些信号必须转换成数字形式才能被计算机处理。通过周期性的对模拟信号抽样进行转换。对每一个样本，模/数（A/D）转换器产生一个 $n$ 位的数表示样本的量级。这个位的数量 $n$ 是根据要求的采样精度来选择的。这些数据送到扬声器时，再使用一个数/模（D/A）转换器将数字信号转换为原来的模拟信号。

抽样过程将产生一个连续的数字化样本流，数字样本的间隔和采样周期同步。这种数据流被称为是等时的，指的是连续的事件被相同的时间段隔开了。

信号必须被足够高的频率抽样，以记下它的最高频率。一般来说，如果抽样频率为每秒 $s$ 个样本，采样过程能够捕捉到最高频率是 $s/2$ 。例如，用8kHz的抽样频率对人说话声音进行采样，就能将声音信息充分捕捉到，声音信号的频率最高可达4kHz。对于更高品质的声音，如在音乐系统中，需要使用更高的采样频率。数字声音的一种标准频率是44.1kHz。每一个样本用4个字节的数据表示以满足高品质声音所需的大音量范围（声强范围）。这将产生大约为1.4Mb/s的数据传输速率。

273 在处理采样的声音和音乐信号时，有一个很重要的要求就是在采样和重放过程中保持精确的时序。高度的抖动（样本时序的变化）是不允许的。因此，计算机和音乐系统之间的数据传送机制必须保持一致的样本间延迟。否则就需要引入复杂的缓冲和重定序电路。偶尔的错误或样本丢失是可以接受的。听者可能完全注意不到，或者只产生不引人注意的滴答声。并不需要复杂的机制来确保完全正确的数据传输。

图像或视频数据的传输也有相似的要求，但传输带宽要高得多。带宽指的是一个通信通道的总数据传输能力，一般采用一个合适的单位，比如用每秒多少比特或字节来度量。为了保证商用电视的图像质量，一张图片需要用160kB来表示，并且每秒钟发送30帧，即44Mb/s的总带宽。在高品质图像中，如HDTV（高清晰度电视）会要求更高的传输速率。

大容量存储设备如硬盘和只读光盘存储器（CD-ROM）的要求与以上内容不同。这些设备是计算机存储器结构的一部分，将在第5章中介绍。它们到计算机的连接必须提供至少40或50Mb/s的数据传输速率。磁盘机制产生的延迟约为1毫秒。因此，向计算机发送或从计算机中接收数据时产生的较小额外延迟可以忽略，也不用考虑抖动。

### 即插即用

随着计算机的普及，它们的形式也应该越来越透明。例如，现在的家庭影院至少包含一台计算机，而人们使用家庭影院时，不应该关闭或重启计算机来连接或断开影院设备。

即插即用特性指的是一台新设备（如附带的扬声器）可以在系统运行时随时连接到计算机上。系统应该能够自动检测到新设备，识别相应的设备驱动软件及所有其他服务这台设备的设施，并分配适当的地址，同时和计算机建立逻辑连接使二者能够互相通信。

即插即用的要求涉及系统的所有层次，从硬件到操作系统和应用软件。USB设计的一个主要目标就是提供即插即用功能。

### USB体系结构

上面的讨论指出了对具有廉价、灵活和高数据传输带宽的互连系统的需求。并且，I/O设备可能与要连接的计算机之间相距较远。对于高带宽需求，通常使用能够并行传送8、16或更多位宽的总线。但是，大量的连线将会增加成本和复杂度，对于用户操作也不方便。而且，由于4.5.2

节中提到的数据相位偏移问题也很难设计出能够将数据传输很远距离的宽总线方式。相位偏移的数量随着距离的增大而增加,这将限制可用的速率。

USB选择的是串行传输格式,因为串行总线具有廉价和灵活性。时钟和数据信息一起被编码,作为单个信号来传输。因此,串行总线没有时钟频率,也不存在由于数据相位偏移导致的距离限制。因此可以通过使用高时钟频率来提供高数据传送带宽。如前面提到的USB提供三种不同的位速率,从1.5Mb/s到480Mb/s,来满足不同的I/O设备。

[274]

为了满足随时添加或删除大量设备的需求,USB使用了图4-43所示的树形结构。每一个树结点都有一个集线器,它是主机和I/O设备之间的中间控制点。在树的根部有一个根集线器将整棵树连接到主计算机上。树的叶子是被服务的I/O设备(如键盘、Internet连接器、扬声器或数字电视),它们在USB术语中称为功能设备。为了与本书后面章节保持一致,将这些设备称为I/O设备。

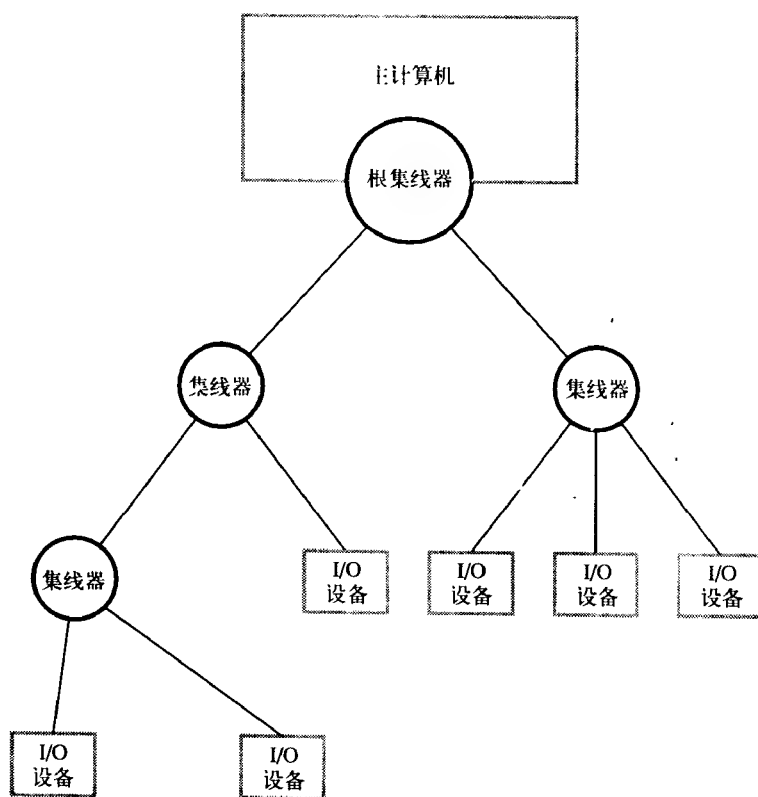


图4-43 通用串行总线的树形结构

树结构保证了仅仅使用简单的点到点串行连接就可以将大量设备连接到计算机上。每一个集线器都有很多连接设备的端口,这些设备还包括其他集线器。在正常的操作中,集线器从上游连接接收消息,复制给所有下游端口。因此,主计算机发布的消息可以广播到所有I/O设备中,但只有被寻址的设备才响应这条消息。就这一方面来说,USB的功能与图4-1中的总线相同。但是,与图4-1不同的是I/O设备的消息只能上行发送到树根,而其他设备不能看见。因此,I/O设备只能与主机进行通信,不能与其他I/O设备进行通信。

[275]

现在来看一下树型结构是如何满足USB设计目标的。树型结构使得可以通过很少的端口(根集线器)就可以将大量的设备连接到计算机上。同时,每台I/O设备通过一个串行的点到点连接

和计算机相连。这是便于实现即插即用特性的一个很重要的思想，后面将简要介绍。并且，由于电气传输的原因，这种连接上的串行数据传输要比图4-1中的总线上的并行传输容易得多。串行传输中，可以使用高数据传输速率和长电缆。

USB的操作是严格基于查询的。设备只有在响应主机查询时才发送消息。因此，上行的消息彼此互不冲突和干扰，因为没有两台设备可以同时发送消息。所以可以采用非常简单和廉价的集线器。

不管是全速还是低速设备都必须遵循上面描述的操作模式。但是，当在USB 2.0版本中引入了高速操作后就必须引入另一种操作模式。考虑图4-44中的情况，集线器A通过一个高速连接器与根集线器相连。这个集线器连接有一台高速设备C和一台低速设备D。通常给设备D的消息是从根集线器低速发送的，速度为1.5Mb/s，因此即使一条很短的消息也需要花费数十微秒。在这条消息期间不能进行其他的数据传送，因此就降低了高速连接的效率，还会产生高速设备不可接受的延迟。为缓和该问题，USB规范规定在高速连接上传送的数据必须总是高速传送的，即使最终接收者是一台低速设备。因此，给设备D的消息将高速地从根集线器发送到集线器A，然后继续低速地传送至设备D。后一段传送将要花费较长的时间，在这期间到其他结点的高速传输可以继续进行。例如，当低速消息从集线器A发送到设备D时，根集线器可能已经与设备C交换了好几条消息。在这段时间里，总线被认为拆分成了高速和低速两种运行。在给设备D消息之前和之后都有专门的指令发送给集线器A，分别来启动和终止拆分传输操作模式。

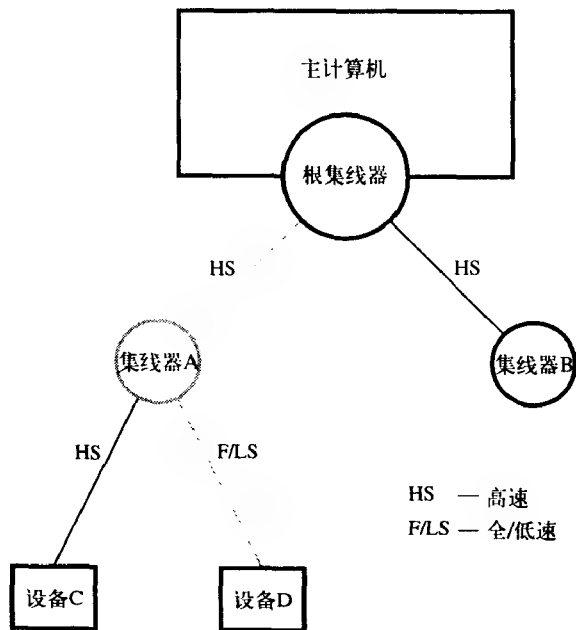


图4-44 拆分总线操作

USB标准规定了USB互连的硬部件细节，以及主机软件的组织和要求。USB软件为应用软件和I/O设备提供双向连接。这些连接称为管道。数据在管道的一端输入，在管道的另一端移交。寻址、时序或错误检测和恢复等问题由USB协议处理。

在4.2.6节提到负责传输发送给或来自I/O设备数据的软件称为设备的驱动程序。设备驱动程序依赖于它支持的设备特性。因此，更准确地说，USB管道是负责连接I/O设备和驱动程序的。当一台设备与计算机相连时，并且USB软件分配给它惟一的地址空间后，管道便被建立起来了。之后，数据随时都可以从这个管道中流过。

下面分析一下在USB总线上设备是如何被寻址的，然后讨论数据传送的各种方式。

#### 寻址

在前面对输入输出操作的讲述中，提到在一般情况下，是通过分配给I/O设备惟一的存储器地址来标识它们的。实际上，一台设备通常有好几个可寻址的地址，来使软件发送和接收控制和状态信息并传送数据。

当USB总线连接到主计算机上时，它的根集线器连接到处理器总线上，看起来就像是一个单

台设备一样。主机软件与连接到USB上的每一台设备通过发送信息包进行通信，根集线器收到信息包后接着将它发送给USB树中的相应设备。

USB上的每一台设备——集线器或I/O设备，都分配有一个7位的地址。这个地址是USB树的局部地址，与处理器总线上使用的地址没有任何关系。一个集线器可以连接任意数量的设备和集线器，地址的分配是任意的。当一台设备第一次连接到集线器上或被启动时，它的地址为0。集线器中连接这台设备的硬件可以检测到该设备已经连接上了，并记下该事实作为自己状态信息的一部分。主机周期地查询每一个集线器来收集状态信息，了解是否有新增或被删除的设备。当主机知道连接了一台新设备后，给相应的集线器端口发送一个复位的指令序列，从设备处读取信息，然后向设备发送配置信息，并为设备分配一个惟一的USB地址。这些动作完成后，设备就开始正常工作并只响应该新地址了。

277

上面描述的初始连接过程是支持USB的即插即用功能的一个关键特性。这个过程完全由主机软件控制。主机软件可以检测到已经连接上的设备，读取有关设备的信息，这些信息通常存储在设备硬件中一个很小的只读存储器中，并发送配置设备的指令来允许和禁止某些特性和功能，最后给设备分配一个惟一的USB地址。惟一需要用户做的是将设备插入到集线器的端口中并打开电源开关。

当一台设备被关闭时，也有一个类似的过程。相应的集线器将这个事实报告给USB系统软件，然后系统软件更新它的设备表。当然，如果断开的设备本身是一个集线器，所有通过该集线器与计算机连接的设备都必须记录为断开。USB软件必须随时保持总线拓扑和所有连接设备的完整映像。

设备中能够进行数据传输的位置，如状态、控制和数据寄存器，称为端点，它由一个4位的编码来识别。实际上，每一个4位的数值标识一对端点，一个用于输入，另一个用于输出。因此，一台设备最多可以有16对输入/输出端点。一个双向数据传送的USB管道与一对端点相连。连接0号端点的管道任何时候都存在，包括设备启动或复位后的瞬间。它是USB软件在启动过程中使用的控制管道。在启动过程中，使用其他端点对的管道也可能被建立，这取决于设备的需要和复杂性。4位端点编码将作为主机寻址信息的一部分被发送，后面会做简要的介绍。

### USB协议

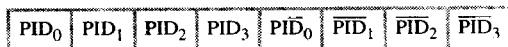
USB上传输的所有信息都被组织成包的形式，一个包由一个或多个字节组成。包的种类很多分别执行不同的功能。我们将通过列举一些主要类型包的例子来解释USB的操作并说明如何使用它们。

USB上传输的信息可以被分成两大类：控制和数据。控制包执行寻址设备启动数据传送、应答数据已经被正确接收和指示一个错误等任务。数据包运载转交给设备的信息。例如，输入和输出数据就是在数据包中传输的。

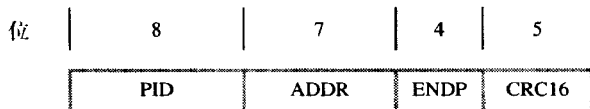
278

一个包由一个或多个包含有不同类型的信息字段组成。第一个字段存放包的识别码PID，它标识包的类型。有4位信息，它们需要传输两次。第一次传送的是它们的真实值，第二次传送的是每一位的补码，如图4-45a所示。这样接收设备能够进行校验，确认是否已经收到了正确的PID值。

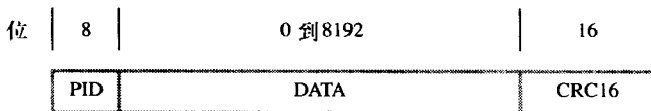
4位PID码可以标识16种包的类型。一些控制包如ACK（应答）只由PID字节组成。用于控制数据传送操作的控制包称为标记包，它的格式如图4-45b所示。标记包从PID域开始，使用两个PID值来分别区分IN信息包和OUT信息包，它们分别控制输入和输出数据传输。PID域的后面是7位设备地址和该设备内的4位端点编号。包中的后5位为错误校验，使用循环冗余校验法（CRC）。CRC位是根据地址和端点域的内容计算出来的。接收设备执行逆运算，确定接收到的包是否正确。



a) 信息包识别字段



b) 包标记为IN或OUT



c) 数据包

图4-45 USB包格式

数据包传输输入和输出数据，格式如图4-45c所示。识别字段之后最多可存放8192位数据，最后是16位错误校验位。有三种不同的PID格式用来标识数据包，因此数据包可以编号为0、1、2这三种。需要注意的是数据包不包含设备地址和端点编号，这些信息包含在启动传送的IN或OUT令牌包中。

考虑一台连接到USB集线器的输出设备，该集线器被连接到主计算机上。图4-46显示了一个输出操作的例子。主计算机发送一个OUT类型的标记包给集线器，接着发送一个包含有输出数据的数据包。数据包的PID域标记它是一个0号数据包。集线器通过校验错误控制位确认传输无误后，发送一个应答包（ACK）给主机。集线器继续将标记和数据包往下传送。所有I/O设备都将会接收到这些包序列，但只有在标记包中识别出自己地址的设备才接受后面包中的数据。确认传送无误后，设备发送ACK包给集线器。

全速或低速管道中的连续数据包交替使用0号和1号类型数据包。这样在发生错误传输时，比较容易恢复。如果一个标记、数据或应答包由于传输错误丢失了，发送者重发整个包序列。通过检查PID区域中的数据包编号，接收者可以检测并丢弃重复包。高速数据包按顺序循环编号为0、1、2、0、…。

输入操作也遵循一个类似的过程。主机发送一个包含设备地址的IN类型的标记包。实际上这是一次查询，要求设备将它现有的输入数据发送给主机。设备发送一个数据包给主机进行响应，然后发送一个ACK包。如果设备没有准备好的数据，它就发送一个否认（NAK）包。

在前面的叙述中，我们指出既有全/低速连接又有高速连接的总线使用拆分传输操作模式，来保证高速连接上的消息不被延迟。在这种情况下，在向全速或低速设备发送IN或OUT包之前有一个专门的控制包启动拆分传输模式。

现在大家应该对USB上的数据传送协议有所了解了。上述事务可以按照不同的方法执行，并有许多协议规则管理设备的行为。有关USB协议更详细的描述请参看USB协议说明书<sup>[3]</sup>。

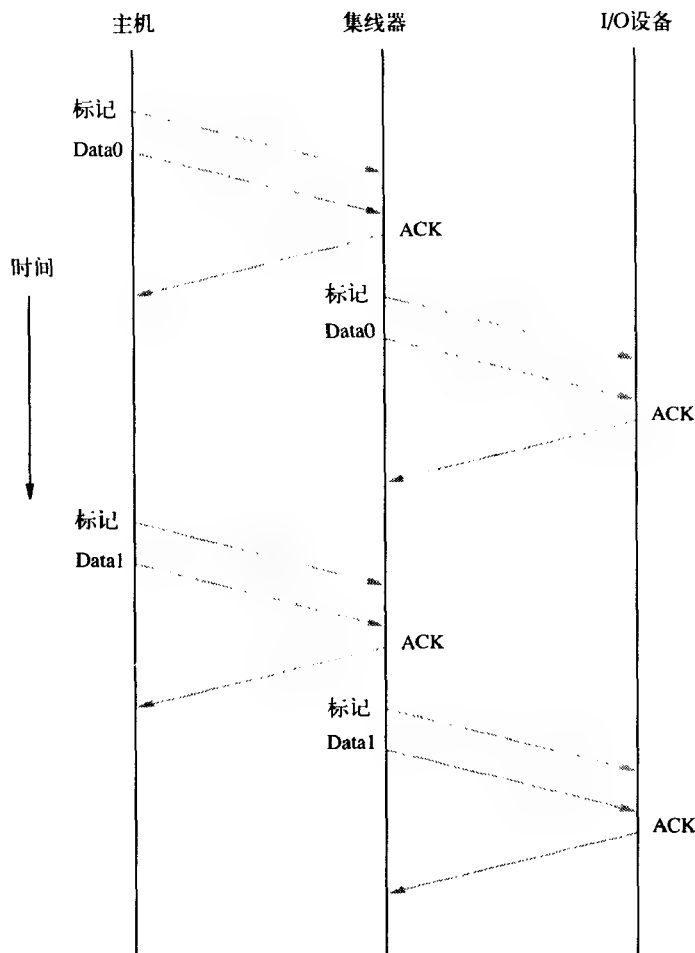


图4-46 一次输出传输

### USB上的等时传输

USB的一个主要目标是支持以简单的方式传输等时数据，如被采样的声音。产生或接收等时数据的设备需要有时间参考来控制采样过程。为了提供这个参考，USB上的数据传输被划分成等长度的帧。对于全速和低速数据一帧就是1ms。根集线器每1ms精确地产生一个起始帧控制包（SOF）来标识新帧的开始。

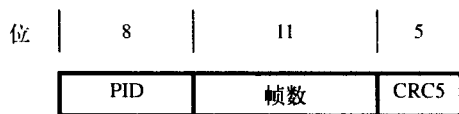
SOF包到达任何设备都可以作为一个标准的时钟信号被设备利用。为了支持需要更长时间周期的设备，SOF包有11位的帧编号，如图4-47a所示。SOF包之后，主机对等时设备进行输入输出传送。这样每台设备每1ms就有一次机会进行输入输出传输。

等时传输的主要要求是一致的时序。偶尔的错误可以接受，因此不需要重发丢失的包或发送应答包。图4-47b显示了紧跟SOF后面的两次传输。包含设备地址3的控制包之后是发送给该设备的数据，输入或输出数据取决于控制包是IN还是OUT类型。这里没有应答包，下一个传输序列是给设备7的。

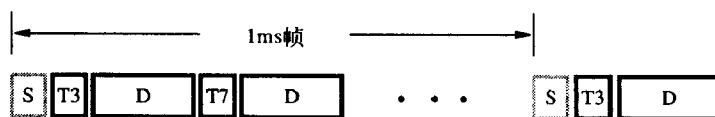
举一个例子，假设给设备3的数据包中包含8个字节的数据。如果等时通道的速率为64kb/s，那么在每一帧中可以发送一个包。这种通道可以用于声音连接。传输8个字节的数据需要3个字节的标记包，其后面紧接着11字节的数据包（包括PID和CRC区域），总共132位。此外，至少还需



要3个字节用于时钟同步和标识包序列结束。在12Mb/s的速度下,将花费13s。很明显,一帧中 can 支持几台这样的设备。在服务完总线上的所有等时设备后,帧中剩余的时间可以用来服务异步设备,交换控制和状态信息。



a) SOF包



S — 帧起始包  
 $Tn$  — 标记包, 地址 =  $n$   
 D — 数据包  
 A — ACK包

b) 帧举例

图4-47 USB的帧

等时数据只允许在全速和高速连接上传输。在高速连接中, SOF包在1ms内等间隔地重复8次形成8个125 $\mu$ s的微帧。

### 电气特性

USB连接使用的电缆由4根电线组成。其中两根用来连接电源 +5V和地。因此, 集线器或I/O设备可能直接从总线获得能源, 也可以使用独立的外部电源。另外两根电线用于传输数据。不同速度的传输使用的信号方式不同。在低速传输中, 通过在两条信号线中的一条上发送高电平(5V), 分别表示发送1和0。在高速连接中使用的是差分传输。

## 4.8 结束语

本章中讲述了三种基本的I/O传输方式。最简单的技术是程序控制输入输出, 在这种技术中处理器在程序指令的直接控制下执行所有需要的控制功能。第二种方式是基于中断的; 这种机制可以中断现有程序的执行, 转去服务具有更高优先级的请求, 这些请求是更需要迫切处理的。虽然所有计算机都有处理这种情况的机制, 但是不同计算机的中断处理系统的复杂程度是不同的。第三种I/O方式是直接存储器访问; DMA控制器在I/O设备和主存之间传送数据, 不需要处理器的连续干预。在这种方式中, DMA控制器和处理器共享对主存的访问。

此外, 还讲述了三种流行的互连标准: PCI、SCSI和USB。它们分别提供不同的方法来满足各种不同设备的需求, 还反映了提高使用方便性的即插即用特性的重要性。

### 习题

4.1 一旦输入数据缓冲区被读取, 接口电路中输入状态位就被清空。为什么要这么做?

- 4.2 写一个程序,在视频显示器上以16进制形式显示主存中10个字节的内容。使用你选择的处理器的汇编指令或伪指令。从存储器LOC位置开始,每个字节用两个16进制字符表示。连续的字符用空格隔开。
- 4.3 计算机有16根地址线,  $A_{15-0}$ 。如果分配给一台设备的地址是  $7CA4_{16}$ , 并且该设备地址译码器忽略  $A_8$  和  $A_9$  两根线。这台设备将对哪些地址进行响应?
- 4.4 子程序与中断服务程序的区别是什么?
- 4.5 在这一章中我们假定只有等到当前机器指令执行完成后中断才被响应。考虑在处理器正在执行一条指令时中断它的操作来响应一个中断的可行性。讨论一下可能产生的问题。
- 4.6 有三台设备A、B、C连接到计算机总线上。所有设备的I/O传送都由中断控制。对于设备A和B不允许中断嵌套,但C的中断请求在服务A或B时可以被接受。在下列情况中给出不同的实现方法:
- (a) 计算机只有一根中断请求线。
  - (b) 计算机有两根中断请求线INTR1和INTR2,INTR1的优先级高一些。
- 说明在每种情况下中断在何时发生,如何被允许和禁止。
- 4.7 一台计算机中有几台设备连接到一根公共中断请求线上,如图4-8a所示。请问你将如何安排,使设备*j*的中断请求在设备*i*的中断服务程序执行完前被接受。尤其当这个系统中有的点上允许中断,有的点上禁止中断时。
- 4.8 考虑图4-8a中的菊花链结构。假设一台设备产生中断请求后,一旦接收到中断应答信号它就将请求撤销。那在进入中断服务程序之前,是否还需要在处理器中禁止中断?为什么?
- 4.9 从一台面向字符的输入设备读取连续的数据块,每个数据块有*N*个字节。程序PROG对每一块数据执行一些计算。分别为68000、ARM和Pentium处理器写一个控制程序CONTROL,执行以下任务:
- (a) 读取数据块1。
  - (b) 启动PROG程序并指向主存中第1块的位置。
  - (c) 当PROG对块1执行计算时,使用中断读取数据块2。
  - (d) 对数据块2启动PROG,同时开始读取块3,如此继续。
- 需要注意的是,CNTRL必须保持正确的缓冲区指针,记录下字符数,并正确地将控制权转交给PROG,不管PROG比数据块输入花费的时间多还是少。
- 4.10 有一台计算机,要求它从20台视频终端接收字符。指针PNTR*n*指向用于存储每一台终端数据的主存区域,*n*从1到20。当另一个程序PROG正在执行时,终端的输入数据必须被收集起来。这可以由如下两种方法实现:
- (a) 每*T*秒钟程序PROG调用一个测试子程序POLL。这个子程序按顺序查询所有20台终端的状态并将所有输入数据传送到存储器,然后返回PROG。
  - (b) 当任何终端的接口缓冲区中有就绪的字符时,产生一个中断请求,然后中断服务程序INTERRUPT被执行。在查询完状态寄存器后,INTERRUPT传送输入字符然后返回PROG。
- 使用伪代码或选择的处理器的汇编语言编写程序POLL和INTERRUPT。假设所有终端的最大字符获取速率为*c*个字符每秒,平均速率等于*rc*,*r*小于等于1。在方法(a)中,能够保证不丢失输入字符的*T*的最大值是多少?在方法(b)中的等价*T*值又是多少?估计一下,在*c* = 100, *r*分别为0.01、0.1、0.5、1时,方法(a)和(b)中服务终端的平均时间百分率是多少。假设

283

284

POLL花费800纳秒查询20台设备，处理设备的中断需要200纳秒。

- 4.11 有一台I/O设备，它使用68000处理器的向量中断功能。

(a) 描述一下当处理器接收到一个中断请求时采取的步骤序列，并给出这些步骤中要求的总线传送的编号。不需要给出总线信号和微程序细节。

(b) 当接收到一个中断请求时，在接受中断前处理器要完成当前指令的执行。分析附录C中的指令表，然后估计一下这期间可能发生的存储器传输的最大数量。

(c) 估计一下从设备请求中断起到中断服务程序的第一条指令被取出执行期间可能出现的总线传输数量。

- 4.12 需要一个逻辑电路来实现图4-8b所示的优先级系统。这个系统处理三条中断请求线。当在INTR<sub>i</sub>接收到一个请求后，系统在INTA<sub>i</sub>线上产生一个应答信号。如果收到的请求多于一个，只响应最高优先级的请求，优先级顺序为

INTR<sub>1</sub>的优先级 > INTR<sub>2</sub>的优先级 > INTR<sub>3</sub>的优先级

(a) 分别给出INTA<sub>1</sub>、INTA<sub>2</sub>和INTA<sub>3</sub>三个输出的真值表。

(b) 给出一个实现该优先级系统的逻辑电路。

(c) 你的设计是否能够方便地扩充到更多的中断请求线？

(d) 通过添加输入DECIDE和RESET，修改你的设计。当输入DECIDE接收到一个脉冲时，INTA<sub>i</sub>被置1，在输入RESET接收到脉冲时被重置为0。

- 4.13 中断和总线仲裁需要一个根据优先级进行选择机制。设计一个电路，有四根输入线，且为REQ<sub>1</sub>到REQ<sub>4</sub>的轮转优先级方式。起初，REQ<sub>1</sub>的优先级最高，REQ<sub>4</sub>的优先级最低。一根请求线被服务后，它的优先级变成最低，下一根请求线获得最高优先级。例如，在REQ<sub>2</sub>被服务后，优先级顺序从高到低依次为REQ<sub>3</sub>、REQ<sub>4</sub>、REQ<sub>1</sub>、REQ<sub>2</sub>。要求有四个输出允许信号GR<sub>1</sub>到GR<sub>4</sub>，每根输入请求线对应一个。当DECIDE线上接收到一个脉冲时其中一根输出线被置位。
- 4.14 68000处理器有一个三线组IPL<sub>2-0</sub>，用于产生中断请求信号。这些线组成一个3位二进制编号，处理器认为它代表优先级最高的中断请求设备。设计一个优先级编码电路，从7台设备接收中断请求，并产生一个3位的代码表示优先级最高的请求。
- 4.15 (本题适合做实验室实验。)假设有一台视频终端连接到你实验室的计算机上，完成如下两个任务。

285

- (a) 写一个按字母表顺序打印字母的I/O程序A。它打印如下的两行，然后停止：

ABC...YZ

ABC...YZ

- (b) 写一个I/O程序B，它按递增顺序打印数字字符0到9三遍。它的输出格式如下：

012...9012...9012...9

将程序A作为主程序，程序B作为中断服务程序，B由在键盘上输入一个字符启动执行。程序B的执行也可以被键盘上输入的另一个字符中断。当程序B完成后，最近被中断的程序在中断点重新开始执行。程序B应该适当的另起一行以使打印输出有如下的格式：

ABC

012...901

012...9012...9012...9

2...9012...9

DE...YZ

另起一行时, 程序需要发送两个字符: CR(0D<sub>16</sub>)和LF(0A<sub>16</sub>)。如何使用处理器优先级来允许和禁止中断嵌套?

- 4.16 (本题适合做实验室实验。)在习题4.15中, 当一个字符序列的打印被中断后重新开始执行时, 字符序列在新行的起始位置接着打印。在这里我们要求添加光标移动控制功能, 保证重新打印字符序列时, 字符在新行中的打印位置与发生中断前打印该字符的位置相同。因此, 打印输出将有如下格式:

ABC

012...901

012...9012...9012...9

2...9012...9

DE...YZ

重新设计你的程序, 使得在中断出现时进入第三个控制程序C。这个程序调用程序B打印数字序列。然后在返回被中断程序前, 发布适当的光标移动指令移动光标。

- 4.17 考虑4.2.5节中描述的断点方式。在断点插入的位置用一条软件中断指令取代程序指令。在返回原始程序之前, 调试程序将原始程序的指令放回原处, 删除断点。请问调试器是如何将原始程序指令放回原处, 执行该指令, 并在其他程序指令执行之前再次插入断点的?

286

- 4.18 ARM的软件中断指令SWI可以被程序用来调用操作系统请求某些服务。请求的服务由这条指令的低8位说明。操作系统提供的每一个服务都由独立的子程序来执行, 这些程序的起始地址存储在一个表中。

(a) 给出一条或多条的指令, 操作系统使用这些指令将SWI指令的低8位复制到寄存器中。

(b) 给出一条或多条指令来调用适当的服务程序。

- 4.19 在某一应用中, 使用集电极开路方式的中断请求线传送的信号是所有相连设备的请求的逻辑“或”。但在另一应用中, 要求这条线产生的信号表示所有连接到总线的设备都准备就绪了。请问如何使用集电极开路方式来实现?

- 4.20 一些计算机中, 只有在中断请求线的前沿处理器才响应。如果有两台相互独立的设备连接到这条线上将会发生什么情况?

- 4.21 图4-20中, 只有当它的总线允许输入线上收到一个由低到高的跳变时, 设备才能成为总线主控。假设设备1请求总线并获得允许。当它正在使用总线时, 设备3启动它的BR输出。画出设备3在设备1释放总线后如何成为总线主控的时序图。

- 4.22 假定在图4-20的总线仲裁机构中, 只要  $\overline{BR}$  有效, 处理器就保持BG1有效。当设备  $i$  请求总线时, 只有在它的BG <sub>$i$</sub>  输入收到一个由低到高跳变时, 才能成为总线主控。

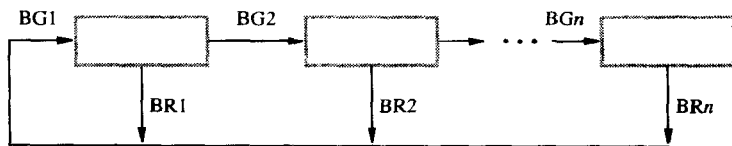
(a) 假设任何时候设备都可以启动BR信号。给出一个事件序列使系统会进入死锁, 死锁时有一台或更多的设备请求总线, 总线也是空闲的, 但没有设备能够成为总线主控。

(b) 采用什么规则来防止发生死锁?

- 4.23 考虑图P4-1的菊花链结构, 总线请求信号作为总线允许信号被直接反馈。假设设备3请求了总线并正在使用总线。当设备3使用完后, 使BR3无效。假设在任何设备中从BG <sub>$i$</sub>  到BG <sub>$(i+1)$</sub>  的延迟都为  $d$ 。证明一个假总线允许脉冲可以从设备3往下传送 (因为该脉冲不是对任何请

287

求的响应, 所以称为“假”)。估计这个脉冲的宽度。



图P4-1 一种分散式总线分配方案

- 4.24 在习题4.23中的设备3释放总线后不久, 设备1和设备5同时请求总线。证明它们都能接收到总线允许信号。
- 4.25 考虑图4-20中的总线仲裁方式。假设在设备接口电路中有一个本地信号BUSREQ, 在设备需要使用总线时置1。设计接口电路中以BUSREQ、BGi 和  $\overline{\text{BBSY}}$  为输入, 并产生  $\overline{\text{BR}}$ 、 $\text{BG}(i+1)$ 、 $\overline{\text{BBSY}}$  为输出的那部分。
- 4.26 考虑图4-22中的总线仲裁电路。假设设备的优先级代码存储在接口电路的一个寄存器中。设计一个电路实现这种仲裁方式。当  $\overline{\text{Start-Arbitration}}$  有效时仲裁开始。稍后, 仲裁电路应该激活在仲裁周期中获胜者的输出线。
- 4.27 如果处理器与I/O设备之间的距离增加了, 图4-26中的时序图将会受到什么影响? 在图4-24中, 如何才能适应距离的增大?
- 4.28 一家工厂使用几个受限传感器来监控温度、压力和其他参数。每个传感器的输出由一个ON/OFF开关组成, 8个这样的传感器都需要连接到一台小型计算机的总线上。设计一个接口, 使8个开关的状态可以作为单个字节从地址FE10<sub>h</sub> 同时读取。假定使用同步总线且使用图4-24中的时序。
- 4.29 设计一个接口在同步总线上连接一个七段显示器作为输出设备。(参见附录A中的图A-37对七段显示器的描述。)
- 4.30 在图4-29的接口中添加中断功能。如何引入一个中断允许位, 其作为接口状态寄存器的第6位可以被处理器置位和清除。当中断被允许且有数据可以被处理器读取时, 接口应该启动中断请求线  $\overline{\text{INTR}}$ 。
- 4.31 处理器总线使用4.5.1节描述的多周期方式。存储单元的速度可以使读操作遵循图4-25中的时序图。设计一个接口电路将存储单元连接到总线上。
- 4.32 考虑总线上的一次写操作, 该总线使用4.5.1节描述的多周期方式。假设处理器在总线事务的第一个周期就能将地址和数据发送完。但这之后存储器还需要两个时钟周期来存储数据。
- (a) 在后两个时钟周期, 总线是否可以用于其他事务?
- (b) 在这种情形中我们是否可以取消存储器的响应? (提示: 仔细分析当处理器试图对存储块进行另一写操作时, 该存储块仍忙于完成前一次请求的情况。请问如何处理这种情况?)
- 4.33 图4-24到图4-26提供了三种不同的总线设计方法。如果被寻址的设备由于故障没有响应, 请问在三种不同方法下会发生什么? 会导致什么问题? 有什么可行的补救方法?
- 4.34 在图4-25中的时序图中, 处理器在总线上保持地址信号直到接收到设备的响应信号。这是必需的吗? 如果处理器只在一个周期内发送地址, 设备端需要什么附加设施?
- 4.35 有一条同步总线按照图4-24中的时序图工作。处理器发送的地址4ns后才能在总线上出现。总线上处理器与不同设备之间的传播延迟在1ns到5ns的范围内, 地址译码需要花费6ns, 被

寻址的设备需要花费5到10ns的时间将请求的数据放到总线上。输入缓冲区需要3ns的准备时间。试问该总线能够工作的最大时钟速度是多少?

4.36 在图4-26中,一次完整的总线传输需要的时间随着延迟不同而不同。有一条和习题4.35中具有相同参数的总线。请问它的最大和最小总线周期是多少?

### 参考文献

1. *PCI Local Bus Specifications*, available at [www.pcisig.com/developers](http://www.pcisig.com/developers).
2. *SCSI-3 Architecture Model (SAM)*, ANSI Standard X3.270, 1996. This and other SCSI documents are available on the web at [www.ansi.org](http://www.ansi.org).
3. *Universal Serial Bus Specification*, available at [www.usb.org/developers](http://www.usb.org/developers).



# 存储器系统

### 本章目标

在本章中你将学习以下内容：

- 基本的存储器电路
- 主存储器的组织结构
- 缩短有效存储器访问时间的高速缓存概念
- 增加主存储器表现容量的虚拟存储器机制
- 用于辅助存储的磁盘、光盘和磁带

291

程序和程序操作的数据保存在计算机的存储器中，在这一章我们将讨论计算机的这个重要部分是如何运作的。到目前为止，读者已经了解到程序的执行速度很大程度上依赖于指令和数据在处理器与存储器之间传输的速度。此外，大容量的内存对于加快处理大量数据的程序执行速度也是十分重要的。

从理想的角度考虑，存储器应该是高速度、大容量、而且很廉价的，但不幸的是不可能同时满足这三个要求，在增加速度和容量的同时会导致成本的增加。为了解决这个问题，人们做了很多工作去开发一些巧妙的结构，这些结构能在合理的成本范围内，提高存储器的表现速度和表现容量。

首先，我们将描述在存储器实现中最常用的组件和组织结构。然后来分析存储器的速度，并且论述如何通过使用高速缓存的方法来提高存储器的表现速度。接着介绍用于增加存储器表现容量的虚拟存储器概念。最后将讨论能提供更大存储能力的辅助存储设备。

### 5.1 基本概念

任何一台计算机中能使用的存储器最大容量取决于寻址方式。例如16位机能产生16位地址，它能在 $2^{16} = 64\text{K}$ 个存储器单元中寻址。类似地，指令能产生32位地址的机器能使用包含 $2^{32} = 4\text{G}$ 个单元的存储器，而40位地址的机器能访问 $2^{40} = 1\text{T}$ 的单元。单元的数目表示计算机地址空间的大小。

大多数现代计算机是按字节编址的，图2-7显示了32位按字节可编址计算机可能的地址分配方案。Big-endian方案用在68000处理器中，little-endian方案用在Intel处理器中。ARM体系结构能够进行配置，可以使用以上任何一种方案。当考虑到存储器结构时，这两种方案之间并没有本质的区别。



存储器通常被设计成按字存储和检索数据,实际上,一次存储器访问存储和检索的位数等于计算机中字长的最常用定义。例如,考虑一台指令能产生32位地址的按字节可编址的计算机,当处理器给存储器发送一个32位地址时,其中高30位的地址决定哪一个字被访问。如果只是访问一个字节,那么最低的两地址指定哪一个字节被访问。在字节读取操作中,其他字节可能也被从存储器中取出,但是它们会被处理器忽略,而在字节写入操作中,存储器控制电路必须保证字中其他字节的内容不被修改。

292 现代计算机存储器的实现非常复杂,刚接触时很难理解。为了简化对存储器结构的介绍,我们首先来看一下传统的体系结构,在后面的部分中再讨论最新的技术方法。

从系统的角度,我们可以把存储器部件看作是一个黑盒子,它通过使用两个寄存器,数据可以在处理器和存储器之间传输,这两个寄存器通常称为MAR(存储器地址寄存器)和MDR(存储器数据寄存器),它们在1.2节中已介绍过。如果MAR的长度是 $k$ 位,MDR的长度是 $n$ 位,那么存储器部件能够包含 $2^k$ 个可编址单元。在一个存储周期中, $n$ 位数据在存储器和处理器中间传输。这个传输在处理器总线上进行,总线包括 $k$ 条地址线和 $n$ 条数据线。总线还包括Read / Write ( $R/\bar{W}$ )控制线和存储器功能完成(MFC)控制线,用于协调数据传输。其他控制线也可以被添加进来,用来指示有多少个字节被传输。图5-1中示意性地给出了处理器和存储器之间的连接。

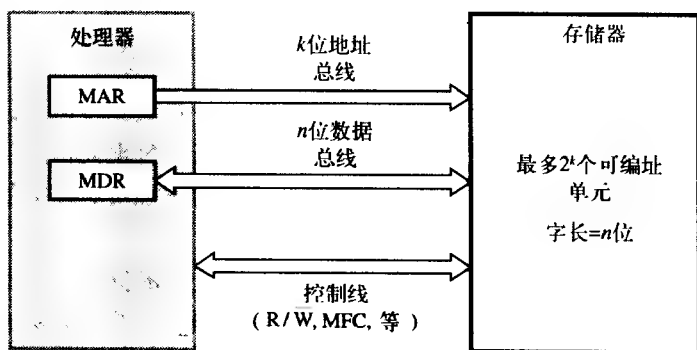


图5-1 存储器到处理器的连接

处理器从存储器读取数据时,把需要读取的存储单元的地址装进MAR寄存器,并把 $R/\bar{W}$ 线设成1。存储器作出响应,把指定位置的数据放到数据线上,并发送MFC信号确认这个操作。当收到MFC信号时,处理器就把数据线上的数据装入MDR寄存器。

处理器写入数据时,把要写入的存储单元地址装入MAR,并把数据装入MDR中,再把 $R/\bar{W}$ 线设成0,表示这是一个写操作。

如果读写操作要访问主存中一个连续的存储单元地址时,可以采用块传输,块传输时只需把块的第一个单元的地址发送到存储器中。在5.5节中我们将会遇到需要块传输的情况。

293 存储器访问可以使用时钟来同步,也可以使用控制总线传输的特殊信号来控制,使用在4.5.1节中描述的总线信号方案。将存储器的读操作和写操作分别看作输入总线传输和输出总线传输来控制。

对存储器速度的一个有用衡量标准是一个操作从开始到结束所用的时间,例如读信号和MFC信号之间的时间,这被称为存储器访问时间。另一个重要的衡量标准是存储器周期时间,它是指相继两个存储器操作开始时刻之间的最小时间延迟,例如相继两个读操作之间的时间。

一般来说,周期时间比访问时间稍长,这与存储器的实现细节有关。

如果对于一个存储器中任何一个存储单元的读写访问都能在固定的时间段内完成,而且这个时间段与存储单元的地址无关,那么这个存储器就称为随机访问存储器(RAM)。这个定义把随机存储器与连续存储设备或部分连续存储设备区分开来,例如磁盘和磁带,后者的访问时间依赖于数据的位置或地址。

实现存储器的基本技术是使用半导体集成电路,后面的章节将介绍关于随机存储器的内部结构和操作的基本情况,然后我们再讨论增加存储器有效速度和容量的技术。

计算机处理器处理指令和数据的速度通常比从存储器中获取指令和数据的速度快,因此,存储器周期时间是整个系统的瓶颈。减少周期时间的一个办法是使用高速缓存。高速缓存是一个容量小、速度快的存储器,它插在容量大、速度慢的主存与处理器之间,保存当前活动的程序段和数据。

虚拟存储器是关于存储器结构的另一个重要概念。到现在为止,我们都是假定处理器产生的地址直接指明物理存储器的存储位置,但并不是所有情况都是这样的。数据在物理存储器中的存储地址可能与程序指定的地址并不一样,具体原因将在后面章节中给出。存储器控制电路把程序指定的地址转换成能用于访问物理存储器的地址,在这种情况下,处理器产生的地址称为虚拟地址或逻辑地址。虚拟地址空间映射到数据实际所在的物理存储器上,映射功能通过一个特殊的存储器控制电路来实现,这个电路通常称为存储器管理部件。映射功能可以根据系统的要求在程序执行时更改。

虚拟存储器用于增加物理存储器的表现容量。数据在虚拟地址空间中编址,虚拟地址空间可以与处理器的寻址能力一样大。但是在任何一个给定时刻,只有虚拟地址空间的活动部分被映射到物理存储器中,而其他部分被映射到使用的大容量存储设备,通常是磁盘中。在程序执行期间当虚拟地址空间的活动部分发生改变时,存储器管理部件更改映射功能,并在磁盘和存储器之间传输数据。这样,在每一个存储器周期,地址处理机制判断在给定地址中的信息是否在物理存储器中。如果在,则相应的字被访问,然后继续执行,否则,从磁盘上将包含要访问字的页传递到物理存储器中,换页过程在5.7.1节中解释。这个页替换当前存储器中某个非活动页。因为在磁盘和存储器之间移动页需要花费时间,所以频繁的页移动会导致访问速度的降低。但是,通过合理地选择哪一个页被换出存储器,可以在相当长的一段时间内使得处理器要访问的字以很高的概率保存在物理存储器中。

[294]

这一节已经简单地介绍了存储器系统的一些组织结构特征,这些特征已经被开发出来,并用来在计算机系统总体成本允许的范围内提供尽可能大和尽可能快的存储器。现在不要求读者了解全部的概念和含义,后面部分将给出更多的细节。我们把这些术语放在一起介绍,使大家理解它们是相互联系的,了解它们之间的关系与对单个特征细节的学习一样重要。

## 5.2 半导体随机存储器

半导体存储器可以在很广泛的速度范围内使用,它们的周期时间可以从100ns到小于10ns。半导体存储器在20世纪60年代刚刚出现时,比它们所替换的磁芯存储器要昂贵得多。由于VLSI(超大规模集成电路)的快速发展,半导体存储器的成本显著地下降。因此,目前在实现内存时,只使用半导体存储器。在这一节,我们将讨论半导体存储器的主要特性。首先介绍在芯片内存器单元的组织方法。

### 5.2.1 存储器芯片的内部组织结构

存储器单元通常按阵列的形式构成, 其中每个单元存储一位 (bit) 信息。图5-2描述了一种可能的组织方式。每一个单元行组成存储器中一个字, 所有的单元行通过一条公共的线连接到一起, 这条线称为字线, 它是由芯片的地址译码器驱动的。每列的单元通过两条位线连接到一个读出/写电路上, 读出/写电路连接到芯片的数据输入/输出线上。在读操作期间, 这些电路读取通过字线选择的单元所存储的信息, 并把这些信息传输到输出数据线上。在写操作期间, 读出/写电路接收输入信息, 并把它们存储到选定的字对应的单元中。

图5-2是一个非常小的存储器芯片例子, 它包含16个字, 每个字包括8位, 这称为 $16 \times 8$ 的结构。每一个读出/写电路的数据输入输出连接到一条双向数据线上, 这个数据线可以连接到计算机的数据总线上。除了地址线 and 数据线以外, 还有两条控制线, 它们分别是 $R/\bar{W}$ 和 $CS$ 。 $R/\bar{W}$ 输入指定请求的操作, 而 $CS$  (芯片选择) 输入在多芯片存储器系统中选择一个给定的芯片, 这些将在5.2.4节中讨论。

295

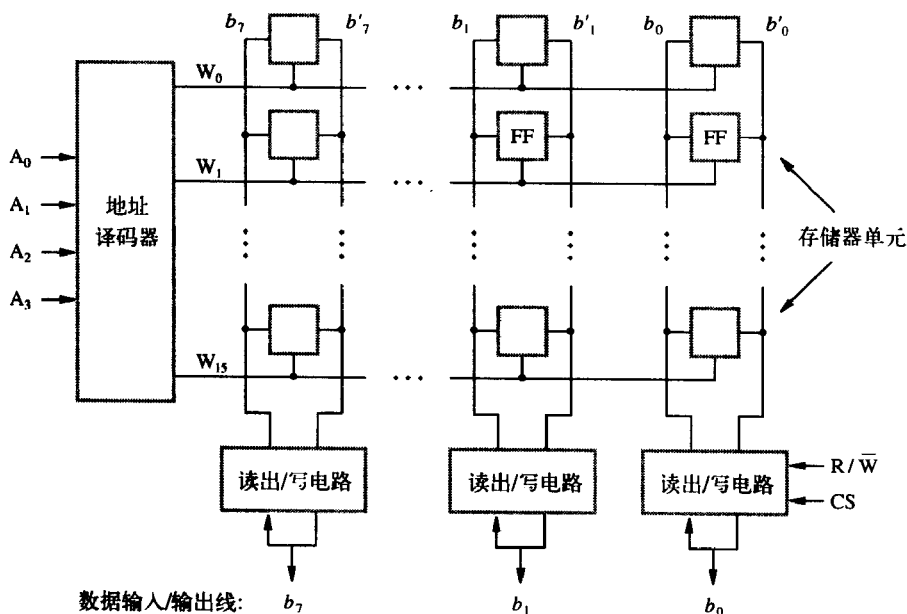


图5-2 存储器芯片内部单元的组织结构

图5-2中的存储器电路存储了128位, 需要为地址、数据和控制提供14条引线。当然, 它还需要两条线用于电源支持和接地。现在来考虑一个稍微大一些的存储器电路, 它包含1K (1024) 个存储单元。这个电路可以被组织成 $128 \times 8$ 的存储器, 总共需要19根引线。换一种方式, 我们可以把这个电路组织成 $1K \times 1$ 的形式。在这种情况下, 需要10位地址, 但只需要一根数据线, 因此共需要15根引线。图5-3显示了这种组织结构。需要的10位地址被分成两组, 每组5位, 分别构成单元阵列的行地址和列地址。行地址选择一个行, 包含32个存储单元, 它们被并行访问。但是, 根据列地址, 这些单元中只有一个通过输入输出多路复用器连接到外部数据线上。

商业上可用的存储器芯片所包含的存储单元数比图5-2和图5-3显示的例子要多得多, 我们使用小的例子可以更容易理解。大规模芯片的组织结构从本质上来说与图5-3所示的一样, 只不过使用更大的存储器单元阵列, 并且有更多的引线。例如, 一个4M的芯片的组织结构可能是512K

× 8，这时需要19根地址输入/输出引脚和8根数据输入/输出引脚。现在容量达数百兆的芯片也已经出现了。

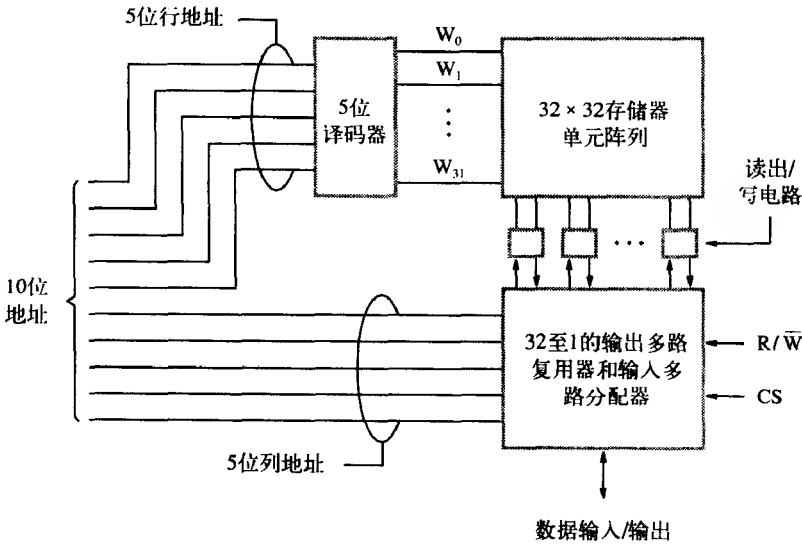


图5-3 1K × 1的存储器芯片组织结构

5.2.2 静态存储器

有的电路只要不停止供电，就能一直保持它的状态，由这样电路组成的存储器称为静态存储器。图5-4描述了如何实现静态随机存储器（SRAM）的存储单元。两个反相器交叉耦合形成一个锁存器，锁存器通过晶体管 $T_1$ 和 $T_2$ 连接到两根位线上，这些晶体管起到开关的作用，可以在字线的控制下打开或关闭。当字线在低电平，晶体管断开，锁存器保持它的状态。例如，我们假设如果X点的逻辑值为1且Y点的逻辑值为0时存储单元处于状态1，只要字线的信号处于低电平，这个状态将一直保持下去。

读操作

为了读取SRAM存储单元的状态，字线被激活以闭合开关 $T_1$ 和 $T_2$ 。如果单元处于状态1，那么位线 $b$ 上的信号是高电平，而位线 $b'$ 上的信号是低电平。如果单元的状态为0，则正好相反。因此， $b$ 和 $b'$ 是互补的。位于位线一端的读出/写电路监测 $b$ 和 $b'$ 的状态，并相应设置输出。

写操作

要设置一个单元的状态时把适当的值放到位线 $b$ 上，并把它的补值放到 $b'$ 上，然后激活字线，这样就迫使单元进入相应的状态。位线上需要的信号由读出/写电路产生。

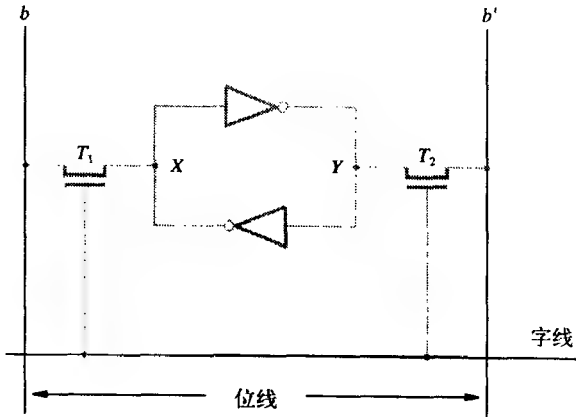


图5-4 静态随机存储器单元

### CMOS单元

实现图5-4中存储单元的CMOS单元在图5-5中给出。晶体管对  $(T_3, T_5)$  和  $(T_4, T_6)$  形成锁存器中的反相器（见附录A）。存储单元的状态就像刚才解释的那样被读写。例如，位于状态1时，通过打开  $T_3$  和  $T_6$  使X点保持在高电平，而  $T_4$  和  $T_5$  是关闭的。这样，如果  $T_1$  和  $T_2$  打开（闭合），位线  $b$  和  $b'$  将分别具有高电平和低电平信号。

298

在老的CMOS静态随机存储器中，电源供应电压  $V_{supply}$  是5V，在新的低压版本中是3.3V。注意，存储单元需要持续的电压供应来维持它的状态，如果电源中断了，单元中的内容就会丢失。当电源恢复时，单元进入一个稳定状态，但这个状态不一定与电源中断前单元的状态相同。因为存储的内容在掉电后会丢失，所以SRAM被称为易失性存储器。

CMOS静态随机存储器的一个主要优点是它们的功耗非常低，因为只有被访问时电流才流入存储单元。此外， $T_1$ 、 $T_2$  和每个反相器中的一个晶体管都是关闭的，这样保证在  $V_{supply}$  和地之间没有有效通路。

静态随机存储器的访问速度非常快，商业芯片中已经有访问时间达几纳秒的芯片。SRAM通常用于速度为关键因素的应用场合中。

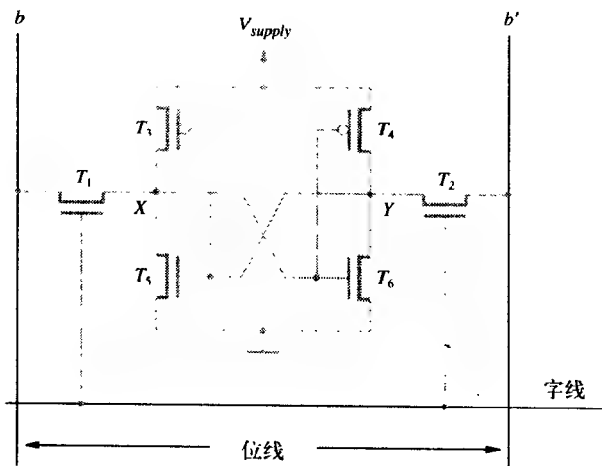


图5-5 CMOS存储器单元的例子

### 5.2.3 异步动态随机存储器

静态随机存储器速度很快，但是它们的成本很高，因为存储单元需要多个晶体管。如果使用更简单的存储单元，那么就可以实现更廉价的随机存储器。但是，这样的单元不能无限期地保存它们的状态，因此被称为动态随机存储器（DRAM）。

信息以电容中电荷的形式保存在动态存储器单元中，这些电荷只能保持10毫秒。但是我们需要存储器单元在更长的时间内保存信息，因此必须通过把电容器中电荷恢复成满值来定期刷新存储单元中的内容。

299

图5-6显示了一个由电容器  $C$  和晶体管  $T$  组成的动态存储器单元的例子。为了在单元中存储信息，晶体管  $T$  打开，一个适当的电压加到位线上，这使得一定数量的电荷存储到电容中。

晶体管关闭后，电容开始放电，这是由于电容本身有泄漏电阻，而且晶体管关闭后，仍会有微弱的电流通过，这个电流只有几皮安。因此只有在电容上的电荷降低到低于某个阈值之前，才能正确读出存储单元中存储的信息。在读操作期间，选中单元的晶体管会打开，连接到位线上的一个传感放大器检测电容上的电荷是否高于某个阈值。如果高于，那么它把位线上的电压提高成满电压以表示逻辑值1。这个电压重新把电容器的电荷充满，使它对逻辑值1。如果传感放大器检测到电容电

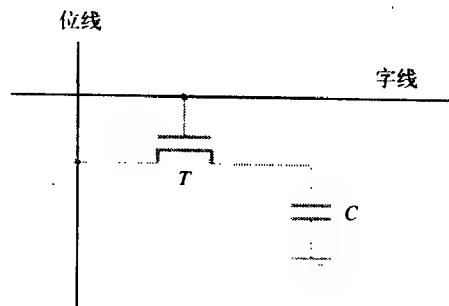


图5-6 单晶体管动态存储器单元

荷低于阈值，它就把位线拉成低电平，保证电容器没有电荷，对应逻辑值0。因而，读取存储单元信息的时候会自动刷新它的内容。选定行中所有的单元被同时读取，这样就会刷新整行的内容。传感放大器电路的实现细节超出了本书的范围。

图5-7显示了一个16M位的DRAM芯片，被配置成 $2M \times 8$ 的形式。存储单元被组织成 $4K \times 4K$ 的阵列，每行的4096个单元被分成512组，每组8个单元，所以一行可以存储512字节的数据。选择一行需要12根地址线，在一个选定的行中指定一个8位的组还需要9根地址线。因此，在这个存储器中访问一个字节需要21根地址线，高12位和低9位分别构成一个字节的行地址和列地址。为了减少外部连接的引脚数，行地址和列地址多路复用12根引脚。在读或写操作期间，行地址首先被加载，芯片响应在行地址选通（RAS）输入线上的脉冲信号，把行地址装入行地址锁存器中。然后读操作开始，选定行上的所有单元被读取和刷新。行地址被装载后，很快列地址被加载到地址引脚上，然后在列地址选通（CAS）信号的控制下，列地址被装入列地址锁存器。列地址锁存器中的信息被译码，该列地址对应的一个8位组的读出/写电路被选中。如果 $R/\bar{W}$ 控制信号指示这是一个读操作，那么选定电路的输出值被传输到数据线 $D_{7-0}$ 上。对于一个写操作， $D_{7-0}$ 上的信息被传输给选定的电路，然后用这些信息覆盖选定的8列存储单元的内容。我们应该注意，在商业DRAM芯片中，RAS和CAS控制信号是低电平有效，因此当它们从高电平变成低电平时，会导致地址被锁存。为了表示这种情况，在图中这些信号表示成 $\overline{RAS}$ 和 $\overline{CAS}$ 。

300

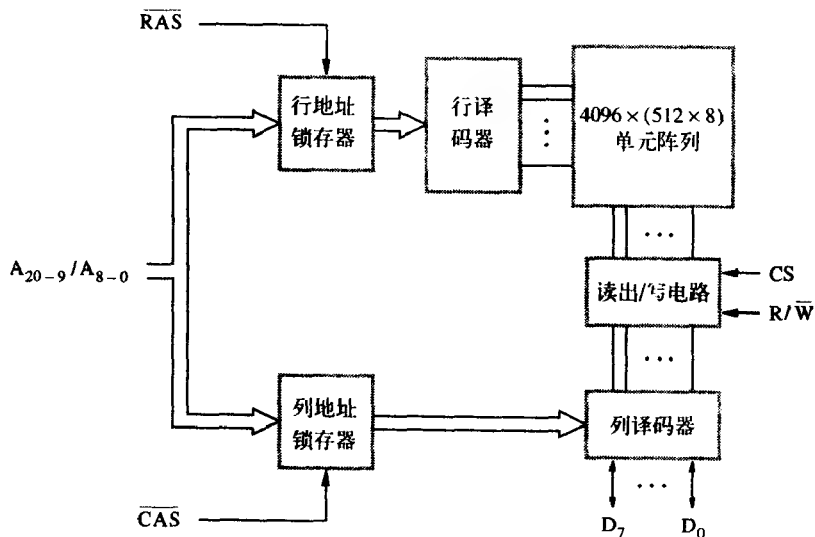


图5-7  $2M \times 8$ 的动态存储器芯片的内部组织结构

不论是读操作还是写操作，加载行地址会导致相应行中的所有单元都被读取并刷新。为了保证DRAM中的内容被保留，必须定期访问每一行中的单元。刷新电路自动完成这个功能。很多动态存储器芯片都在芯片内部自带一个刷新装置，这时，这些存储器芯片的动态属性对于用户来说几乎是看不出来的。

在本节描述的DRAM中，存储器设备的时序是异步控制的。一个特定的存储控制器电路提供控制信号RAS和CAS，它负责控制时序，处理器必须考虑存储器响应的延时。这种存储器称为异步动态随机存储器（asynchronous DRAM）。

由于DRAM的高密度和低成本，它被广泛应用于计算机的存储器部件中。可用芯片的容量

从1M到256M,甚至会开发出更大容量的芯片。为了减少计算机需要的存储器芯片的数量,DRAM芯片被组织成可以并行读写多个位,如图5-7所示。为了在设计存储器系统时提供更多的灵活性,这些芯片以多种组织形式生产,例如,一个64M位的芯片可以组织成 $16\text{M} \times 4$ 、 $8\text{M} \times 8$ 或 $4\text{M} \times 16$ 的形式。

### 快速页模式

当图5-7所示的DRAM被访问时,选中行的4096个单元的内容都被读取,但是只有8位信息被放在数据线 $D_{7-0}$ 上,这个字节通过列地址位 $A_{8-0}$ 来选择。可以做一个简单的改进,使得在访问同一行中其他字节的时候不需要重新进行行选择。在每行传感放大器的输出端加上一个锁存器,加载行地址的时候,会把选定行中的每个位装入相应的锁存器中。于是,只要加载不同的列地址就能把其他的字节放到数据线上。

[301]

最有用的方式是按顺序传输字节,这需要在连续CAS信号控制下加载连续的列地址序列。这种方案在传输一块数据的时候要比使用随机地址传送的速率快得多。这种块传输能力称为快速页模式。(通常行业用语中把少量字节称为块,而把大批字节称为页。)

按规则模式访问内存的应用程序可以利用在块传输时获得的高速率特性,例如图形终端。这个特征在通用计算机的主存和高速缓存之间的数据块传送时也非常有用,我们将在5.5节中解释这一点。

### 5.2.4 同步动态随机存储器

随着近来存储器技术的发展,出现了操作直接与时钟信号同步的动态随机存储器,这种存储器称为同步动态随机存储器(SDRAM)。图5-8显示了SDRAM的结构。它的单元阵列与异步动态随机存储器一样,地址和数据连接通过寄存器进行缓冲。我们应该特别注意在每个传感放大器的输出端连有一个锁存器,读操作使得选中行中所有单元的内容被装入锁存器。但是,如果只是为了刷新而访问,那么锁存器中的内容不会改变,它只会刷新存储单元中的内容。保存在选中列对应锁存器中的数据被传输到数据输出寄存器中,然后就可以在数据输出引脚上获得这些数据。

SDRAM有几种不同的操作模式,可以通过向模式寄存器中写入控制信息来选择这些模式,例如可以指定不同长度的脉冲串操作。使用块传输能力的脉冲串操作,就是上面描述的快速页模式特征。在SDRAM中,不需要在CAS线上提供外部产生的脉冲去选择连续的列,芯片内部使用列计数器和时钟信号来提供需要的控制信号。新数据在每个时钟周期内被放到数据线上,所有的操作由时钟信号的上升沿触发。

[302]

图5-9是一个典型的长度为4的读脉冲串时序图。首先,行地址在 $\overline{\text{RAS}}$ 信号的控制下被锁存。存储器通常使用2到3个时钟周期(在图中我们使用两个)激活选中的行。然后,列地址在 $\overline{\text{CAS}}$ 信号的控制下被锁存。一个时钟周期后,第一组数据被放到数据线上。然后SDRAM自动增加列地址去访问选定行中后面的三组数据,在随后的三个时钟周期内把这三组数据放到数据线上。

SDRAM有内置的刷新电路,其中一个部分是刷新计数器,提供要刷新的行地址。在一个典型的SDRAM中,每一行至少要每64ms刷新一次。

商业SDRAM能够在超过100MHz的时钟频率下使用,这些芯片是为满足大量使用的商用处理器的需求而设计的。例如,Intel制定了PC100和PC133总线标准,分别规定系统总线(主存与之连接)受100MHz和133MHz的时钟控制,因此各主要存储器芯片生产厂商都生产100MHz和133MHz的SDRAM芯片。

[303]

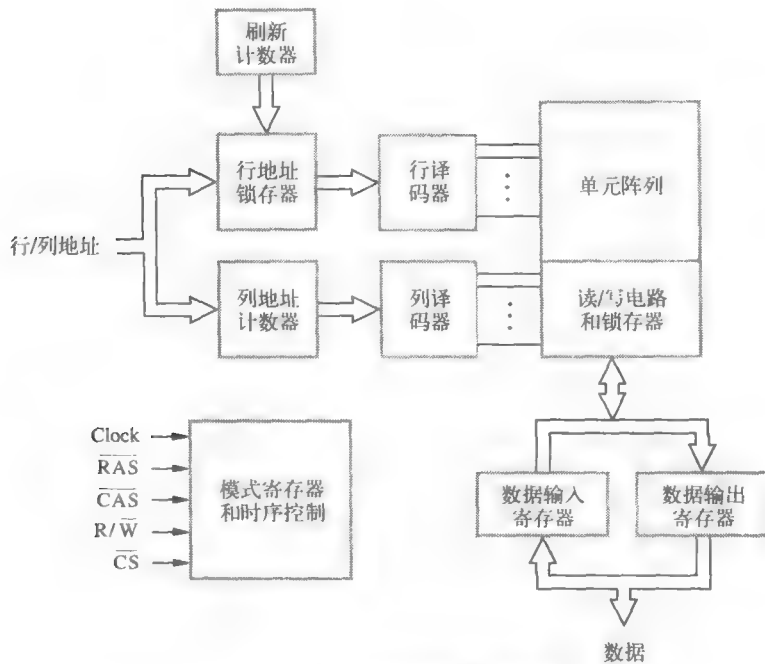


图5-8 同步动态随机存储器

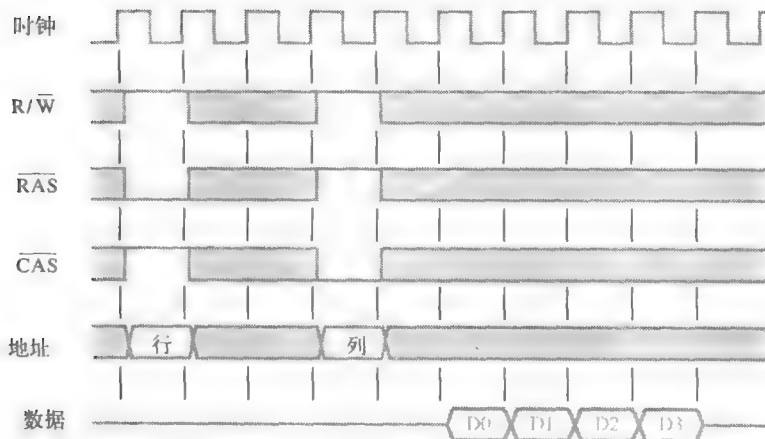


图5-9 在SDRAM中长度为4的读脉冲串

### 延迟与带宽

存储器和处理器之间的传输包括传输单个字或传输一小块字（传输送至或来自于处理器高速缓存，高速缓存在5.5节讨论）。大的块由一页数据组成，它在存储器与磁盘之间传输，这在第5.7节中描述。这些传输的速度和效率对计算机系统的性能影响很大，一个合理的性能指标由两个参数给出：延迟和带宽。

存储器延迟是指向或从存储器传输一个字的数据需要花费的时间。当读写单个字的数据时，延迟完全可以用来标志存储器的性能，但是对于传输一块数据的脉冲串操作，完成操作的时间还依赖于后继字节的传输速度以及块的大小。在块传输中，延迟用来表示传输数据的第一个字



所花费的时间,这个时间通常比传输块中后继的字所花费的时间长得多。例如,在图5-9所示的时序图中,当  $\overline{\text{RAS}}$  信号出现时,访问周期开始,第一个字在五个时钟周期后传输,因此延迟是五个时钟周期。如果时钟频率是100MHz,那么延迟是50ns。剩下的三个字在后续的时钟周期中传输。

当传输数据块时,知道需要多长时间完成整个块的传输是很重要的。因为块的大小各不相同,所以根据在一秒钟之内能传输的位或字节的数量来定义性能度量标准是很有用的,这个标准通常称为存储器带宽。存储器部件(由一个或多个存储器芯片构成)的带宽依赖于访问存储数据的速度和能并行访问的位数。但是,计算机系统(包括存储器和处理器之间的数据传输)的有效带宽不仅仅由存储器的速度决定,它还与连接的存储器及与处理器连线的传输能力有关,一般就是指总线速度。存储器芯片通常按通用总线的速度需求设计。显然,带宽与访问速度、沿单线的传输速度及并行访问位数,也就是连线的数目有关,因此,带宽等于数据传输(访问)速率与数据总线宽度的乘积。

### 双倍数据速率SDRAM

通过对提高性能的不断探索,SDRAM的更快版本被开发出来。标准SDRAM在时钟信号的上升沿执行所有的动作,而现在有一种类似的存储器设备出现了,它以同样的方式访问存储器单元阵列,但是它在时钟信号的上升沿和下降沿都传输数据。这种设备的延迟与标准SDRAM一样,但是由于在时钟信号的两个边沿传输数据,因此在脉冲串传输时,它们的实际带宽是原来的两倍。这种设备就是双倍数据速率SDRAM(double-data-rate SDRAM, DDR SDRAM)。

为了能以足够高的速率访问数据,存储器单元阵列被组织成两个存储体,每个存储体都能单独访问。一个给定块中连续的字被存储到不同的存储体中,这种字的交叉可以实现同时访问在一个时钟信号的连续边沿上传输的两个字。我们将在5.6.1节更详细地考虑交叉的概念。

DDR SDRAM和标准SDRAM在经常使用块传输的应用中很有效,在主存主要向或从处理器高速缓存传输数据的通用计算机中也是这样,参看5.5节。在高质量视频播放时也需要块传输。

## 5.2.5 大容量存储器的结构

我们已经讨论了存储器电路的基本组织结构,它们可以在一个单独的芯片上实现,下面来分析存储器芯片如何连接到一起形成更大的存储器。

### 静态存储器系统

考虑一个由2M(2 097 152)个32位字组成的存储器,图5-10显示了如何使用512K×8的静态存储器芯片来实现这个存储器。图中每一列包括4个芯片,实现了一个字节的位置,四列合起来提供了需要的2M×32的存储器。每一个芯片有一个控制输入,称为芯片选择(Chip Select),当这个输入被设成1时,允许芯片从数据线上接收数据或把数据放到数据线上。每个芯片的数据输出都是三态的(参见A.5.4节),只有选中的芯片能把数据放到数据输出线上,所有其他芯片的输出处于高阻抗状态。地址的高2位被译码,用来决定4个芯片选择控制信号中哪一个被激活,其余的19个地址位用来在选中行的每个芯片内指定哪个字节位置被访问。所有芯片的 $\text{R}/\overline{\text{W}}$ 输入连到一起,提供一个通用的 $\text{R}/\overline{\text{W}}$ 控制(没有在图中显示)。

### 动态存储器系统

大规模动态存储器系统的组织结构本质上与图5-10中的存储器一样,但是物理实现通常更方便,它使用存储器模块的形式。

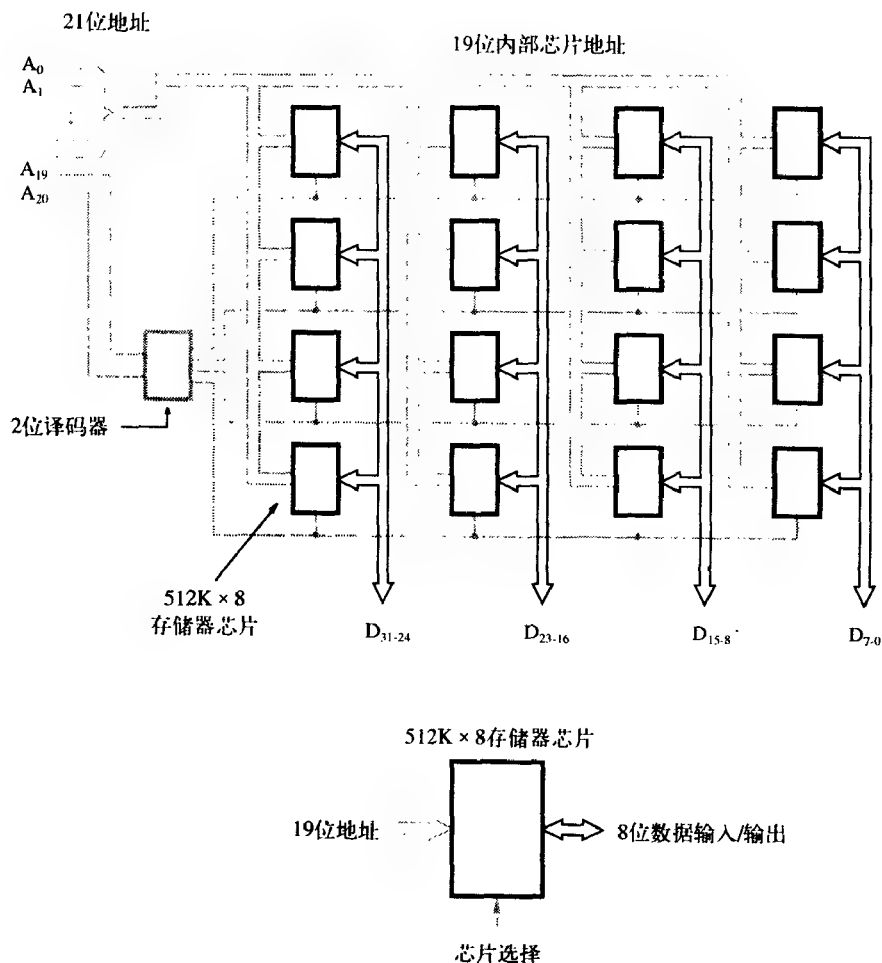


图5-10 使用512K × 8静态存储器芯片的2M × 32存储器模块的组织结构

现代计算机使用非常大的存储器，即使是小型个人计算机也可能最少有32M内存，典型的工作站至少有128M内存。大容量的内存会带来更好的性能，因为更多当前处理的程序和数据能被保存在内存中，这样能降低访问辅助存储设备的频率。但是，如果通过把DRAM芯片直接放到含有处理器的主系统印刷电路板，也就是主板上的方法来建立大容量内存的话，它将占用板上很大的空间，这是无法接受的。而且这样很难支持将来内存的扩展，因为必须要为预计的最大容量分配空间和布线。这些因素导致了更大的存储器部件的发展，这就是所谓的SIMM（单列直插存储器模块，Single In-line Memory Module）和DIMM（双列直插存储器模块，Dual In-line Memory Module）。这种模块把一些存储器芯片组装到一个独立的小板子上，然后把它垂直插入主板上一个单独的槽中。不同容量的SIMM和DIMM被设计成使用相同大小的槽，例如4M × 32位、16M × 32位和32M × 32位的DIMM都使用100针的槽。类似地，8M × 64位、16M × 64位、32M × 64位和64M × 72位的DIMM使用168针的槽。这种模块只占用主板上很小的一块空间，并且很容易通过把原模块替换成使用相同的槽但容量更大的模块来实现扩展。

### 5.2.6 存储器系统因素

为给定的应用选择随机存储器芯片要考虑多个因素，其中最重要的是成本、速度、功耗和

芯片容量。

静态随机存储器芯片一般只用于高速操作的场合，它们的成本和容量受实现基本单元电路的复杂程度的影响。静态随机存储器大部分用于高速缓存。动态随机存储器是实现计算机主存的主要选择，这些芯片可以达到很高的密度，使经济地实现大容量存储器变得切实可行。

### 存储器控制器

为了减少引脚的数量，动态随机存储器使用了多路复用的地址输入。地址被分成两个部分，高地址位用来选择存储单元阵列的行，它们首先被提供，并在RAS信号的控制下锁存在存储器芯片中。低地址位用来选择列，它们随后通过相同的引脚提供，并使用CAS信号锁存。

典型的处理器同时发出所有的地址位，使用到的地址多路复用功能通常由存储器控制器电路完成，如图5-11所示，它位于处理器和动态随机存储器之间。控制器在请求信号的控制下从处理器接收全部的地址和 $R/\bar{W}$ 信号，请求信号表示需要一个存储器访问操作。然后控制器把行地址和列地址传给存储器，并产生 $\overline{RAS}$ 和 $\overline{CAS}$ 信号。这样，控制器除了能完成多路复用的功能外，还能提供RAS-CAS时序。此外它还把 $R/\bar{W}$ 和CS信号送给存储器。CS信号一般是低电平有效，因此在图5-11中它被表示成 $\overline{CS}$ 。处理器和存储器之间的数据线是直接连接的。需要注意的是SDRAM芯片需要时钟信号。

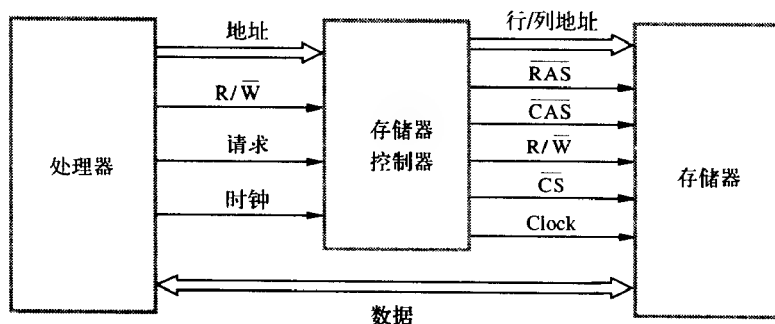


图5-11 存储器控制器的使用

当使用没有自刷新功能的DRAM芯片时，存储器控制器必须提供控制刷新过程的所有信息。这包括一个刷新计数器，用来提供下一行的地址。它的功能是在特定设备指定的周期内使所有行都被刷新。

### 刷新开销

所有的动态存储器都必须被刷新。在原来的DRAM中，刷新所有行的典型周期是16ms，而在一般的SDRAM中，周期是64ms。

考虑一个存储单元排列成8K (= 8192) 行的SDRAM，假设访问一行需要4个时钟周期，那么它刷新所有行需要 $8192 \times 4 = 32768$ 个时钟周期。在133MHz的时钟频率下，刷新所有行需要的时间是 $32768 / (133 \times 10^6) = 246 \times 10^{-6}$ 秒。于是刷新过程在每64ms间隔内占用0.246毫秒，因此刷新开销是 $0.246 / 64 = 0.0038$ ，它小于访问存储器可用总时间的0.4%。

### 5.2.7 Rambus存储器

动态随机存储器的性能由它的延迟和带宽来表示。由于所有的动态随机存储器芯片使用类似的存储单元阵列组织结构，如果它们使用相同的生产过程产生，那么延迟将趋于一致。另一

方面,存储器系统的有效带宽不仅仅依赖于存储器芯片的结构,还依赖于连接到处理器的连线性质。DDR SDRAM和标准SDRAM都是连接到处理器总线上,因此传输速度不只是看存储器设备的速度,还要看总线的速度。时钟频率为133MHz的总线最快每7.5ns传输一次,如果时钟信号的两个边沿都使用的话,则传输两次。在有速度限制的总线上增加传输数据能力的惟一办法就是增加数据线,但是这样会导致总线变宽。

过宽的总线成本很高,并且需要占用主板上很大的空间。另一个可用的办法是实现一个较窄的但是速度很快的总线。Rambus公司采用了这种办法,他们开发了一种自己的设计方案,称为Rambus。Rambus技术的关键是在芯片之间传输信息时使用了一种快速发送信号的方法。它没有采用使用电压0和 $V_{supply}$ 表示逻辑值的信号,它的信号由与参考电压 $V_{ref}$ 有微小偏差的电压组成。参考电压大约是2伏,两个逻辑值用与参考电压上下偏差0.3伏的电压表示。这种类型的信号一般称为微分信号。小的电压浮动占用较短的传输时间,因此可以实现更快的传输速度。

微分信号和高传输速率需要在设计通讯连接线时使用特殊的技术,这种需求增加了拓宽总线的难度。它还需要为处理微分信号设计特别的电路接口。Rambus提供了设计这种线路的完整规范,称为Rambus通道。现在的Rambus设计允许400MHz的时钟频率,此外,数据在时钟信号的两个边沿上传输,因此有效数据传输速率是800MHz。

[308]

Rambus需要特别设计的芯片,这些芯片使用基于标准DRAM技术的存储单元阵列,使用多体单元阵列同时访问多个字,与Rambus通道交互需要的电路包含在芯片内部。这种芯片称为Rambus动态随机存储器(RDRAM)。

Rambus最初的标准提供了一个由9条数据线、一些控制线和电源线组成的通道,其中8条数据线用于传输一个字节的数,第9根数据线可以用于奇偶校验等用途。后来的标准允许附加的通道。两通道Rambus称为直接RDRAM,它有18条数据线,能同时传输两字节数据,没有单独的地址线。

在处理器或其他作为主设备的设备与作为从设备的RDRAM模块之间的通讯,通过在数据线上传输信息包的方法来实现。有三种类型的信息包,分别是请求、响应和数据。由主设备发出的请求包指明要执行的操作类型,它包含期望的存储器位置的地址,还包括一个8位的计数器,用来指定要传输多少个字。操作类型包括存储器读和写,还包括读写RDRAM芯片的各个控制寄存器。当主设备发出请求包后,如果地址选定的从设备能立刻满足这个请求,那么它发回一个肯定的响应包作为回应,否则发回一个否定的响应包表示正忙,这时主设备将会重试。

请求包的位数超过了数据线的条数,这意味着需要几个时钟周期才能传送整个包。窄的通讯连接可以用高传输速率来弥补。

RDRAM芯片可以组装成更大的模块,类似于SIMM和DIMM。一个这样的模块称为RIMM,它能支持16个RDRAM。

Rambus技术直接与DDR SDRAM技术竞争,它们各自有各自的优点和缺陷。一个非技术因素是,DDR SDRAM的规范是一个开放的标准,而RDRAM是Rambus公司私有的设计方案,芯片生产商必须为此付使用费。最后我们应该记住,在存储器市场,假设性能都能满足要求,那么组件的价格往往是决定性因素。

### 5.3 只读存储器

SRAM和DRAM芯片都是易失的,这意味着当电源关闭后,它们存储的信息将会丢失。有

很多应用要求存储器芯片在电源关闭后仍能保留它所存储的信息。例如，在典型的计算机中，硬盘驱动器用来保存大量的信息，包括操作系统软件。当计算机开机时，必须把操作系统从硬盘中装入内存，这需要执行一个引导操作系统的程序。由于引导程序很大，它的大部分都存储在磁盘上，处理器必须执行一些指令把引导程序装入内存。如果整个存储器都是由易失的存储器芯片构成的，那么处理器将没有办法访问这些指令。一种实用的解决办法是提供一小块非易失性的存储器来保存从磁盘装入引导程序的指令。

非易失存储器广泛应用于嵌入式系统，这将在第9章中介绍。这种系统通常不使用磁盘存储设备，它们的程序存储在非易失的半导体存储器设备中。

现在已经开发了不同类型的非易失存储器，一般可以像使用SRAM和DRAM一样读取它们的内容。但是，把信息存入这些存储器需要特殊的写过程。因为它们一般的操作只涉及读取存储数据，因此这种类型的存储器称为只读存储器（ROM）。

### 5.3.1 ROM

图5-12显示了一种可能的ROM单元结构，如果晶体管在P点接地，那么单元存储的逻辑值为0，否则为1。位线通过一个电阻连到电源线上。要读取单元的状态，需要激活字线，这样晶体管开关闭合。如果晶体管接地的话，位线上的电压会接近0电位；如果没有接地的话，位线上的电压仍然保持高电位，表示逻辑值1。位线末端的传感电路产生正确的输出值。数据是在ROM制造时被写入的。

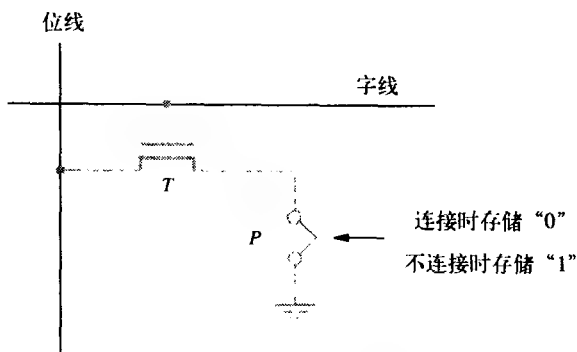


图5-12 只读存储器单元

有些ROM设计成由用户装入数据，即可编程只读存储器（PROM）。可编程能力是通过在图5-12中的P点插入一根熔丝实现的。在编程之前，存储器包含的内容全是0，用户可以通过使用高电流脉冲把所需位置的熔丝烧断，使那些位置的值为1。当然，这个过程是不可逆的。

PROM提供了ROM不具备的灵活性和方便性。后者在存储固定的程序和数据且大量生产ROM的情况下，具有很大的经济吸引力。但是，如果只需要很少数量，为准备存储特定信息需要的掩码所付出的成本使得它们非常昂贵。在这种情况下，PROM提供了一种更快而且非常廉价的方法，因为它们可以直接由用户编程。

### 5.3.3 EPROM

另一种ROM芯片允许擦除存储的数据，并装入新的数据，这种可擦除、可再编程的ROM通常称为可擦除可编程只读存储器（EPROM），它在数字系统的开发阶段提供了相当大的灵活性。因为EPROM能在很长的时间内保存数据，所以当软件正在被开发的时候，它们可以替换ROM。通过这种方法，可以很容易地实现存储信息的改变和升级。

EPROM存储单元的结构与图5-12中的ROM单元相似。但是，在EPROM的单元中，P点总是接地的，而且它使用特殊的晶体管，这个晶体管既可以当作普通晶体管用，也可以当作总是关

闭状态的无效晶体管用。可以对晶体管编程使它成为一个始终打开的开关，这种方式可以通过向它注入电荷来实现，注入的电荷被内部捕获。这样，EPROM存储单元可以用前面讨论的ROM单元所用的方法来构建存储器。

EPROM芯片的主要优势是可以对它的内容进行擦除和重新编程。擦除需要驱散存储单元中的晶体管捕获的电荷，这可以通过把芯片暴露在紫外线下照射来完成。由于这个原因，EPROM芯片被安装在有透明窗口的外壳中。

### 5.3.4 EEPROM

EPROM的一个重要不足是重新编程时芯片必须从电路上物理移除，而且被紫外线照射时它的所有内容都会被擦除。可以实现另一版本的可擦除PROM，它可以用电来擦除和编程，这样的芯片叫做电可擦除可编程只读存储器（EEPROM），擦除时不需要把它从电路上移除，还可以有选择地擦除存储单元中的内容。EEPROM惟一的不足是擦除、写数据和读数据需要用不同的电压。

[311]

### 5.3.5 闪存

最近，一种与EEPROM技术类似的解决办法出现了，它就是闪存（flash memory）设备。闪存单元的基础是一个被捕获电荷控制的单独晶体管，就像EEPROM一样。虽然有些方面类似，但是闪存和EEPROM还是有本质上的不同。在EEPROM中可以读取和写入单个单元中的内容，而在闪存中可以读取单个单元的内容，但只能写入整块单元的内容。在写之前，这个块以前存储的内容被擦除。闪存有更高的密度，这可以带来更高的容量和更低的成本。它们只需要单一的电压，而且在操作中消耗更少的能量。

低功耗使得闪存在由电池驱动的便携式设备应用中有很强的吸引力。典型的应用包括手持计算机、蜂窝电话、数码相机和MP3音乐播放器。在手持计算机和蜂窝电话中，闪存保存操作设备需要的软件，这样就免除了对磁盘的需求。在数码相机中，闪存用来保存相片的图像数据。在MP3播放器中，闪存存储表现声音的数据。蜂窝电话、数码相机和MP3播放器是很好的嵌入式系统的例子，这些将在第9章详细讨论。

单独的闪存芯片不能为上面提到的应用提供足够的存储容量，它们需要由多个芯片组成更大的存储器模块。实现这样的模块时有两种常见的选择：闪存卡和闪存驱动器。

#### 闪存卡

构建更大模块的一个方法是把闪存芯片安装到一个小卡片上，这样的闪存卡有一个标准的接口，它们可以用在不同的产品上。卡片只需要简单地插入到一个可以很方便地接触到的槽上。闪存卡有不同的容量，典型的容量是8M、32M和64M字节。使用MP3编码格式时，一分钟音乐可以存储在大约1M字节的存储器中，因此，一个64M的闪存卡可以存储一小时的音乐。

#### 闪存驱动器

更大容量的闪存模块已经被开发出来，可以替换硬盘驱动器，这些闪存驱动器完全仿效硬盘设计，它们可以装入标准的磁盘驱动器架中。但是闪存驱动器的存储容量太低了，目前闪存的容量小于1G字节。相反地，硬盘能存储很多G字节。

[312]

闪存驱动器是固态电子设备，没有可动的部分，这为它提供了一些重要的优点。它们有更短的搜索和访问时间，从而有更短的响应时间（搜索和访问时间在5.9节介绍磁盘时讨论）。它们

有更低的功耗，使其在电池驱动的应用中有很大的吸引力，而且受震动的影响很小。

与硬盘驱动器相比，闪存驱动器的缺点是它们的容量比较小，而且每位的成本比较高。硬盘每位的成本非常低。闪存的另一个不足是多次写入后它会退化，幸运的是这个次数很大，一般至少是一百万次。

## 5.4 速度、容量和成本

我们已经说过，理想的存储器应该是高速度、大容量和低价格的。从5.2节的讨论中已经清楚地了解到用SRAM芯片可以实现非常快的存储器，但是由于它们的基本单元有六个晶体管，不能在一个芯片上安放大量的单元，所以很昂贵。因此，出于成本原因，使用SRAM芯片构建大存储器是不现实的。另一个可选的办法是使用DRAM芯片，它们有非常简单的基本单元，因此也便宜很多，但是这样的存储器也要慢得多。

虽然动态存储器部件可以用合理的成本实现数百兆字节的容量，但是可提供的容量与有庞大数据的大程序要求相比还是很小的。一个解决办法是使用辅助存储设备来实现大的存储器空间，这主要是磁盘。以合理的价格可以得到容量非常大的磁盘，而且它们在计算机系统中被广泛地使用，但要比半导体存储器慢得多。所以我们可以得出结论，划算的海量存储可以由磁盘来提供。一个可以提供的大容量主存使用动态随机存储器来构建。静态随机存储器用在更小的部件中，这些部件中速度极其重要，例如高速缓存。

所有这些不同类型的存储器部件都有效地在计算机中使用，计算机的整个存储器可以看作在图5-13中描述的分层结构。最快的访问是对保存在处理器寄存器中数据的访问，因此，如果把处理器寄存器看作存储器层次结构的一部分的话，那么就访问速度而言，处理器寄存器位于最顶端。当然，寄存器只提供了所需存储器的一个很小部分。

在层次结构中的下一层是相对来说容量较小的存储器，它们可以直接在处理器内部实现。这个存储器称为处理器高速缓存，用来保存存储在外部更大存储器中的指令和数据的拷贝。高速缓存概念在图1-6中引入，在5.5节中有详细分析。通常有两级高速缓存。主高速缓存总是放在处理器芯片内部，这个高速缓存很小，因为它要抢占处理器芯片的空间，而这个空间还要实现很多其他的功能。主高速缓存称为一级（L1）高速缓存。一个更大的辅助高速缓存放在主高速缓存和其余存储器之间，这称为二级（L2）高速缓存，它通常用静态随机存储器芯片来实现。

在处理器芯片中包含一个主高速缓存，再使用一个更大的芯片外的二级高速缓存，这是当前设计计算机最通用的方法。但是，

在现实中还能找到其他的设计，可能在处理器芯片上根本没有高速缓存，也可能把一级和二级

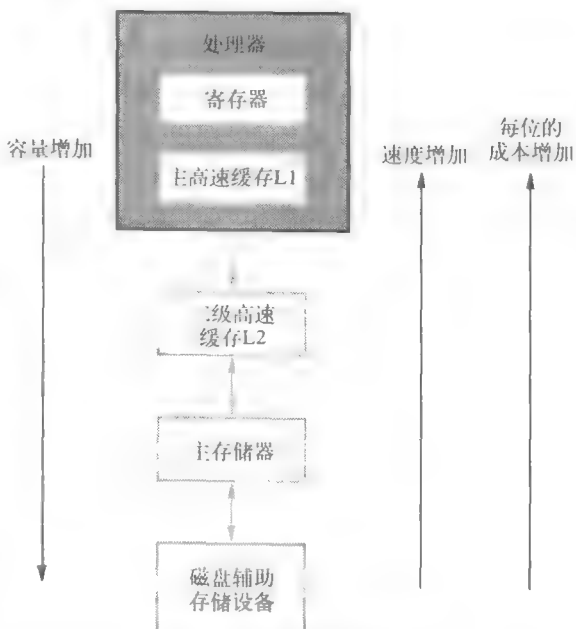


图5-13 存储器层次结构

高速缓存都放到处理器芯片中。

层次结构的再往下一层称为主存储器，这是用动态存储器部件实现的一个很大的存储器，通常采用SIMM、DIMM或RIMM的形式。主存很大，但是比高速缓存要慢得多。在一个典型的计算机中，主存的访问时间大约比一级高速缓存的访问时间长十倍。

磁盘设备提供廉价的大容量存储，与用于主存的半导体设备相比，它们是非常慢的。我们将在5.9节中讨论磁盘技术。

在程序执行期间，存储器访问速度是至关重要的。管理图5-13中存储器分层系统操作的关键是把最近要使用的指令和数据放到离处理器尽可能近的地方，这可以使用随后章节描述的机制来完成。我们首先来详细讨论一下高速缓存。

## 5.5 高速缓存

与现代的处理器相比，主存的速度是非常低的。为了获得更高的效能，处理器不能花费时间去等待从主存中获取指令和数据。因此，设计一种能减少访问所需信息需要时间的方案是很重要的。因为主存部件的速度受电气约束和封装约束的限制，所以解决办法必须不同的体系结构设计寻求。一个有效的解决办法是使用一个快速的高速缓存，这实质上是使主存对处理器而言比实际上表现得更快一些。

314

高速缓存机制的有效性是以计算机程序的引用局部性特征为基础的。对程序的分析显示它们大部分的执行时间花在过程中，其中有很多指令被重复执行。这些指令可能包括一个简单的循环、嵌套的循环或一些反复互相调用的过程。指令序列的具体模式并不重要，关键是在一段时间内程序局部区域的很多指令被反复执行，而程序的其他部分相对来说很少被访问，这称为引用局部性。它表现在两个方面：时间和空间上。时间方面意味着最近执行的指令可能很快又被执行到，而空间方面意味着与最近执行指令邻近（就指令地址而言）的指令也可能很快被执行到。

如果程序的活动段能被放到高速缓存中，那么总的执行时间就会大幅减少。从概念上说，高速缓存的操作非常简单。存储器控制电路利用引用局部性特征而设计。引用局部性的时间方面指出只要信息项（指令或数据）是最先需要的，那么它应该被放入高速缓存，在高速缓存中可能被保留直到需要它。空间方面指出不要每次只从主存中取一项放到高速缓存中，取邻近地址的多个项是很有用的。我们将用术语块来指明一定大小的一组邻近地址位置。另一个经常用来说明高速缓存块的术语是高速缓存块。

考虑图5-14中简单的布局。当从处理器接收到一个读请求时，包含指定单元的一块存储器字的内容被传到高速缓存，每次传一个字。

然后，当程序引用这个块中任何单元时，所需内容直接从高速缓存中读取。通常高速缓存存在给定时刻能保存合理数量的块，但这个块数与主存中所有的块数相比还是要小得多。主存块与高速缓存块的对应关系由映射功能指定。当高速缓存已经满了，而且高速缓存

外的一个存储器字（指令或数据）被引用时，高速缓存控制硬件必须决定移出哪一个块，为包含被引用字的新块腾出空间。做出这种决定的规则集合构成了替换算法。

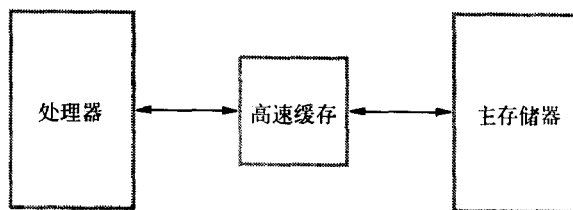


图5-14 高速缓存的使用

315



处理器不需要知道高速缓存的存在,它简单地使用指向内存单元的地址发出读写请求。高速缓存控制电路判断请求的字当前是否存在于高速缓存中。如果在,那么读写操作直接在恰当的高速缓存位置执行,这时称为读写命中。在读操作中并不涉及主存。在写操作中,系统可以按两种方法处理。第一种技术称为直接写协议,高速缓存单元和主存单元同时更新。第二种技术只更新高速缓存单元,并更新与该单元关联的标志位,这个标志位称为脏位或修改位。主存单元以后修改,它的修改是在当包含这种标记字的块为新块腾出空间而被移出高速缓存时进行的。这个技术称为写回协议。直接写协议很简单,但是当一个给定的高速缓存字在高速缓存期间被多次更新时,直接写会造成不必要的对主存写操作。注意写回协议也可能造成不必要的写操作,因为当一个高速缓存块被写回到主存时,块中所有的字都被写回,即使这个块在高速缓存中只有一个字被修改过。

当读操作地址对应的字没有的高速缓存中的时候,发生一次读失效。这时,包含所请求字的一个块从主存中读进高速缓存。整个块装进高速缓存后,请求的这个字传送给处理器。一种替代的方法是字一旦从主存中读出,就立刻送到处理器。后一种方法称为直接装入或早重启,它在一定程度上减少了处理器等待的时间,但是需要更复杂的电路,成本更高。

在写操作期间,如果地址对应的字不在高速缓存中,就发生一次写失效。此时,如果使用直接写方式,那么信息直接写入主存。如果是写回方式,那么该地址对应字的块首先被装入高速缓存,然后高速缓存中对应的字被新信息覆盖。

### 5.5.1 映射功能

为了讨论用于指定存储器块被放入高速缓存哪个位置的方法,我们使用一个很小的例子。  
[316] 考虑一个包含128个块的高速缓存,每个块有16个字,总容量是2048(2K)个字。假设主存用16位地址编址。主存有64K的字,我们把它看作4K个16个字的块。为简便起见,假设连续的地址对应连续的字。

#### 直接映射

决定主存块在高速缓存中位置的最简单方法是直接映射技术。在这项技术中,主存块 $j$ 映射到高速缓存的模128的块 $j$ 上,如图5-15中所示。这样,只要是第0、128、256、...个主存块装入高速缓存,它们都存储在高速缓存的第0块,依次类推。因为多个主存块被映射到指定的一个高速缓存块,所以即使高速缓存没有满,在该单元上也可能引起冲突。例如,一个程序的指令可能从第1块开始,接着可能在一个转移语句后跳到第129块。当这个程序执行时,这两个块都被传输到高速缓存的第1块。可以通过允许新块覆盖已存在的块来解决这种冲突,但是这样的替换价值不大。

高速缓存块的放置由主存地址决定,主存地址可以分成三个字段,如图5-15所示。低4位从16个字的块中选择一个字。当新块进入高速缓存的时候,7位的高速缓存块字段决定这个块存入高速缓存中哪个位置。块的主存地址的高5位存在5个标志位中,与高速缓存块中的位置关联,它们标志映射到这个单元的32个块中哪一个当前正在高速缓存中。在执行过程中,由处理器产生的每个地址的7位高速缓存块字段指向高速缓存中特定的单元。地址的高5位和与高速缓存单元关联的标志位做比较。如果匹配,期望的字就在那个高速缓存块中;如果不匹配,包含期望字的块必须首先从主存中读出并装入高速缓存中。直接映射技术很容易实现,但是不是很灵活。

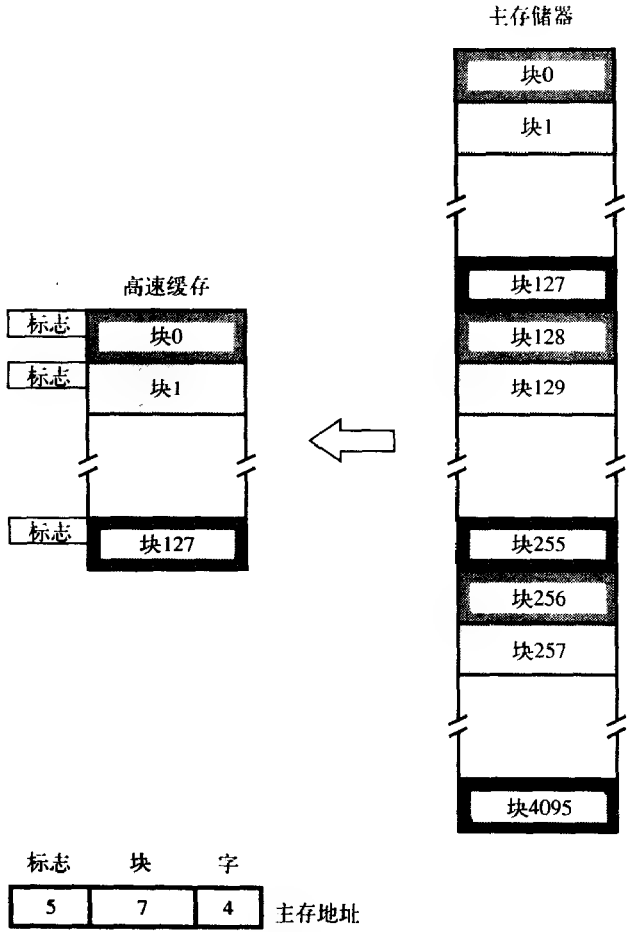


图5-15 直接映射高速缓存

**相联映射**

图5-16给出了一种更灵活的映射方法，通过它可以把主存块放置到高速缓存中任何一个位置上。在这种情况下，当一个主存块在高速缓存中的时候，需要12位标志去标记它。从处理器接收的地址标志位与高速缓存每个块的标志位做比较，看需要的块是否存在。这称为相联映射技术，它可以完全自由地选择在哪个高速缓存位置上放置主存块。这样，高速缓存空间可以得到更有效的利用。只有在高速缓存满了，又需要放入高速缓存块时才必须从高速缓存中替换出一个已经存在的块。在这种情况下，需要一个算法选择出哪个块将要被换出。可以有多种算法进行选择，我们将在5.5.2节中讨论。相联映射高速缓存的成本比直接映射高速缓存要高，因为它要检索所有128个标志的值，判断给定的块是否在高速缓存中。这种检索称为相联检索。由于性能的原因，标志必须并行地被检索。

**组相联映射**

直接映射技术和相联映射技术可以结合起来使用。高速缓存的块被分成组，这种映射允许主存块可以位于特定组中的任何一个块中。这样，放置块时可以有多个选择，从而使直接映射方法的冲突问题得到缓解。同时，通过减少相联检索的大小可以降低硬件成本。图5-17给出了一个组相联映射技术的例子，高速缓存中每个组有两个块。这样，第0、64、128…个主存块映

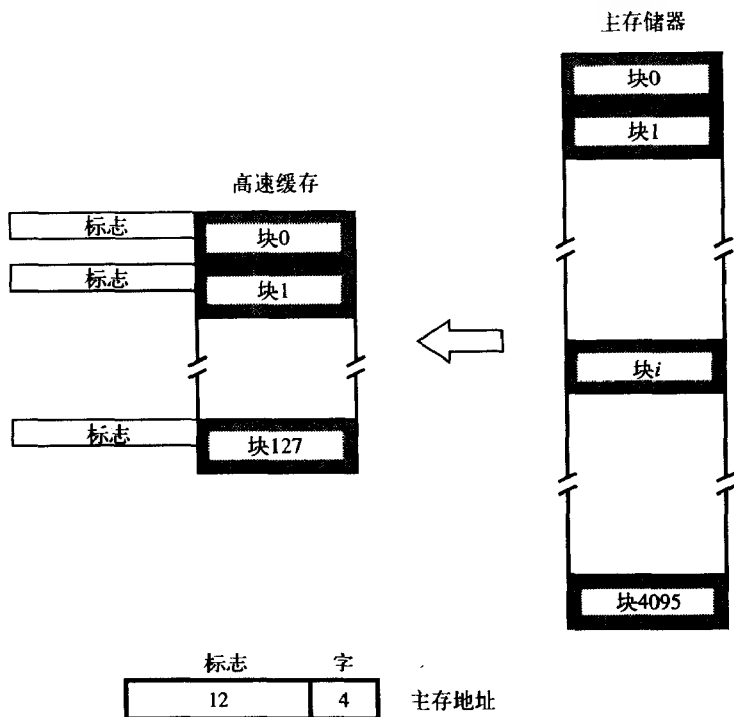


图5-16 相联映射高速缓存

射到高速缓存的第0组，它们可以占用这个组的两个位置中的任意一个。有64个组时，就意味着地址的6位组字段决定高速缓存中的哪个组可能包含期望的块。地址的标志字段与这个组两个块的标志做比较，检查所需要的块是否存在。这两种方法相结合的检索方法实现起来很简单。

每组的块数可以作为一个参数，根据特定计算机的需求进行选择。对于图5-17中的主存和高速缓存的容量来说，5位的组字段可以满足每组4个块，4位的组字段可以满足每组8个块，依次类推。每组128个块的极端情况不需要组字段，它对应完全相联技术，有12个标志位。另一个极端情况是每组只有一个块，这就是直接映射方法。每组有 $k$ 个块的高速缓存称为 $k$ 路组相联高速缓存。

每个块还需要提供另一个控制位，称为有效位，这个位表示这个块是否包含有效数据。不应该把它与前面提到的脏位或修改位混淆起来。脏位表示一个块在高速缓存中时是否被修改过，它只有在系统不使用直接写方法时才需要。当系统电源刚接通或主存从磁盘装入新的程序和数

据时，所有的有效位都被置成0。从磁盘到主存的传输使用DMA机制实现。通常，出于成本和性能考虑，它们绕过高速缓存。当特定的高速缓存块第一次从磁盘上装入时，它的有效位设成1。只要主存块被不经过高速缓存的数据源更新时，就需要做一次检测来判断装载的块当前是否在高速缓存中。如果在，那么它的有效位就被清成0。这样就能保证高速缓存中没有过时的数据。

当完成一个从主存到磁盘的DMA传输并且高速缓存使用写回协议时，会出现一个类似的难题。这时，主存中的数据可能不能反映出对高速缓存中的备份所进行的修改。这个问题的一个解决方法就是转储清除高速缓存，在DMA传输开始之前迫使脏数据写回到主存中。操作系统可以很容易做到这一点，而且这对性能的影响并不大，因为这样的磁盘传输发生的频率不高。保证不同实体（这里指处理器和DMA子系统）使用相同数据备份的需求称为高速缓存一致性问题。

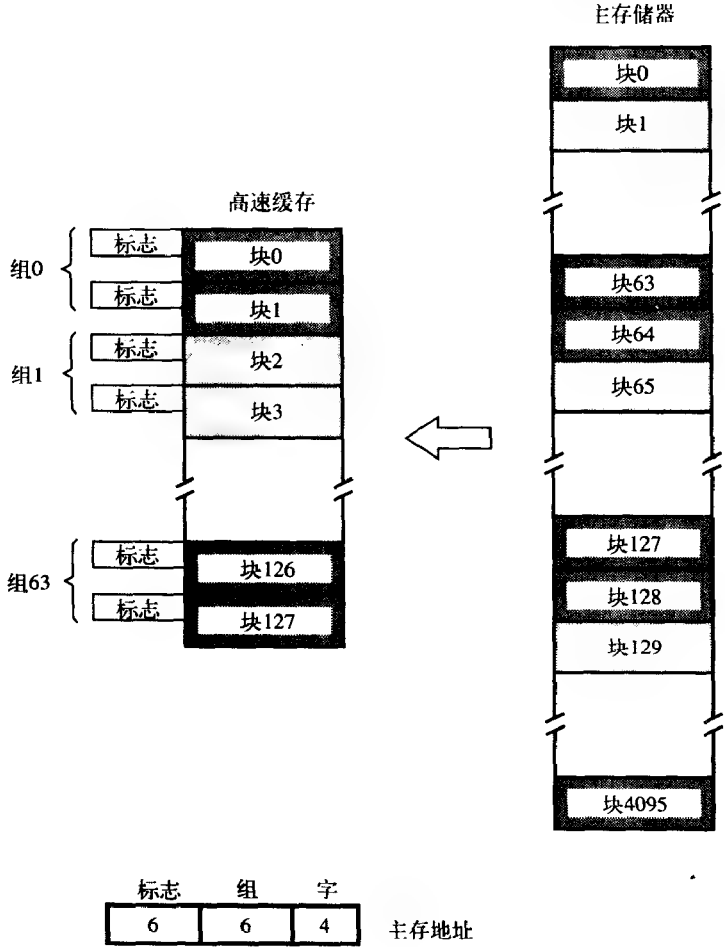


图5-17 每组两个块的组相联映射高速缓存

5.5.2 替换算法

在直接映射高速缓存中，每个块的位置是事先决定的，因此不存在替换策略。在相联和组相联高速缓存中存在一定的灵活性。在一个新块需要放入高速缓存而高速缓存中所有单元都已经满了的情况下，高速缓存控制器必须决定哪一个原有的块将被覆盖。这是一个很重要的问题，因为这个决定是系统性能的决定性因素。通常，我们的目标是保留那些最近可能被引用的高速缓存块。但是决定哪一个高速缓存块将被引用并不是一件容易的事。程序的引用局部性特征为找出一个合理的策略提供了线索。因为程序通常在一个局部区域内保留相当长的时间，所以最近引用的块有很高的概率再次被引用。因此，如果有一个块要被覆盖，那么覆盖最长时间没有被访问的那个块是比较合理的。这个块称为最近最少使用（LRU）块，这项技术称为LRU替换算法。

要使用LRU算法，高速缓存控制器必须在进行计算时跟踪对各个块的引用。假设在每组四块的组相联高速缓存中跟踪LRU块，可以在每个块上使用一个2位的计数器。命中时，被引用块的计数器被设成0，原来比被引用块的计数器值低的计数器都增加1，其他的保持不变。当发生缺块且组不满时，与从主存中装入的新块相联的计数器被设成0，其余块的计数器值增1。当发

生缺块且组已经满了时,计数器值为3的块被移出,新块放到被移出块的位置上,它的计数器设成0,其他三个块计数器增1。很容易证明被占用块的计数器值总是不同的。

LRU算法被广泛地使用。虽然它在很多访问模式下表现得很好,但是在某些情况中会导致很差的性能。例如,在顺序访问一个比较大,无法全部装入高速缓存的数组元素时,它将产生令人失望的结果(参见5.5.3节和习题5.12)。可以用在决定哪个块被替换时引入少量随机性来提高LRU算法的性能。

321 实践中也使用一些其他的替换算法。直觉上一个合理的规则应该是装入新块时移出一个已经满了的组中最旧的块。但是,因为这个算法没有考虑最近访问高速缓存块的模式,所以在选择最合适的块被移出时通常不如LRU算法有效。最简单的算法是随机选择一个块覆盖,有趣的是,人们发现这个简单的算法在实际中非常有效。

### 5.5.3 映射技术的例子

现在来考虑一个详细的例子,说明不同映射技术的效果。假设处理器有单独的指令高速缓存和数据高速缓存。为了使例子简单,假设数据高速缓存只有容纳8块数据的空间,同时假设每块由一个16位字的数据组成,存储器是按字可编址的,它有16位地址(这些参数在实际计算机中并不真实,但是这可以让我们更清楚地描述映射技术)。最后,假设高速缓存在块替换时使用LRU替换算法。

让我们来分析一下运行下面程序引起的数据高速缓存入口的变化: A是 $4 \times 10$ 的数字数组,每个元素占一个字,它存储在单元7A00到7A27处(16进制)。A数组的元素按列顺序存储,如图5-18所示。这个图还说明了不同高速缓存映射技术的标志是如何从存储地址中得到的。注意这里不需要像图5-15到图5-17那样用一些位来标志块中的字,因为我们已经假设每个块中只有一个字。程序用A的行元素的平均值来标准化第一行元素。因此,我们需要计算这行元素的平均值,然后用平均值去除每个元素。需要的任务可以表示为:

$$322 \quad A(0, i) \leftarrow \frac{A(0, i)}{\left(\sum_{j=0}^9 A(0, j)\right)/10}, \quad i = 0, 1, \dots, 9$$

图5-19给出了对应这项任务的程序结构。在这个程序的机器语言实现中,数组元素被编址成存储器单元。我们使用变量SUM和AVE来分别保存总数和平均值。这些变量,还有变址变量*i*和*j*,在计算时都被保存在处理器的寄存器中。

#### 直接映射高速缓存

323 在一个直接映射的数据高速缓存中,高速缓存内容的变化如图5-20所示。表的各列表示图5-19中程序的两个循环各次执行完成后高速缓存中的内容。例如,第一个循环执行完两次后(*j*=1),高速缓存中保存元素A(0, 0)和A(0, 1)。这些元素在块的0和4单元中,由地址的最低3位决定。在下次循环中,元素A(0, 0)被A(0, 2)替换, A(0, 2)也被映射到相同的位置。注意,需要的元素只被映射到高速缓存的两个单元中,其他的六个单元无论在执行标准化任务之前内容是什么,都保持不变。

第一个循环执行十次后(*j*=9),高速缓存中的内容是A(0, 8)和A(0, 9)。由于第二个循环颠倒了元素的处理顺序,头两次循环时将在高速缓存中找到需要的数据。当*i*=7时, A(0, 9)被A(0, 7)替换;当*i*=6时, A(0, 8)被A(0, 6)替换,依次类推。因此在执行第二个循环时,八个元素都被替换了。

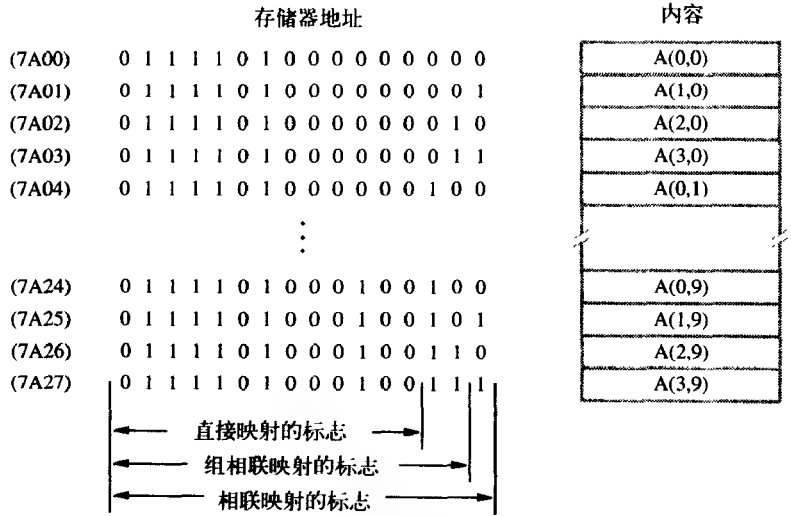


图5-18 存储在主存中的一个数组

```
SUM := 0
for j:= 0 to 9 do
    SUM := SUM + A(0,j)
end
AVE := SUM / 10
for i:= 9 down to 0 do
    A(0,i) := A(0,i) / AVE
end
```

图5-19 5.5.3节中任务的示例

每次循环后数据高速缓存中的内容									
块位置	$j = 1$	$j = 3$	$j = 5$	$j = 7$	$j = 9$	$i = 6$	$i = 4$	$i = 2$	$i = 0$
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

图5-20 直接映射数据高速缓存的内容

读者应该记住为高速缓存中的每个块保存标志，由于空间的因素，我们没有在图中显示它们。  
**相联映射高速缓存**

图5-21显示了相联映射高速缓存时的变化。在第一个循环的头八次循环中，假设高速缓存最一开始是空的，那么元素被放入连续单元的块中。在第9次循环中，LRU算法选择 A (0, 0) 被 A (0, 8) 覆盖。*j* 循环的下一，也是最后一次循环中 A (0, 1) 被 A (0, 9) 替换。现在，第二个循环的头八次循环 (*i* = 9, 8, ..., 2) 都只需要高速缓存中的元素。当 *i* = 1时，需要的元素是 A (0, 1)，所

以它替换最近最少使用的元素  $A(0, 9)$ 。在最后一次循环中,  $A(0, 0)$  替换  $A(0, 8)$ 。

在这种情况下, 当第二个循环执行时, 只有两个元素不能在高速缓存中找到。在直接映射的情况下, 第二个元素执行期间有八个元素需要重新装入。显然, 相联高速缓存受益于可以完全自由地把主存块映射到高速缓存中的任何位置。高速缓存的这种良好应用还因为在程序的第二个循环中我们选择逆序处理元素。如果第二个循环处理元素的顺序与第一个循环一样的话将会怎样呢? 考虑这个问题很有意思。使用LRU算法, 在第二个循环中, 所有元素在使用前都会被覆盖。如果使用随机替换算法, 性能的退化就不会发生。

### 组相联映射高速缓存

在这个例子中, 假设组相联数据高速缓存组织成两个组, 每组能保存四个块。这样, 地址的最低位决定相应的存储器块映射到哪个组中, 高15位组成标志。

图5-22描述了高速缓存内容的变化。因为所有的块都是偶数的地址, 它们映射到第0组。注意在这种情况下, 在第二个循环执行期间, 有六个元素必须被重新装入。

虽然这是一个简单的例子, 但是它说明了一般情况下, 相联映射性能最好, 组相联映射次之, 直接映射最差。但是相联映射实现代价太高, 所以组相联映射是一种很好的折中实用方法。

### 5.5.4 商用处理器中高速缓存的例子

我们现在来考虑在68040、ARM710T、Pentium II和Pentium 4处理器中高速缓存的实现。

#### 68040高速缓存

Motorola 68040在处理器芯片上包含两个高速缓存, 一个用于指令, 另一个用于数据。每个高速缓存有4K字节的容量, 使用图5-23描述的4路组相联结构。这个高速缓存有64个组, 每个组能保存4个块。每个块有4个长字, 每个长字有4个字节。为了映射的目的, 地址按照图中所示的那样进行解释。最低的4位指定块中一个字节的位置, 下面的6位在64个组中确定一个组, 高22位组成标志。为了压缩图中的符号表示, 每个字段中的内容用16进制显示。

高速缓存控制机制中包括每块有一个有效位以及块中的每个长字脏位, 这些位已在5.5.1节中解释过。当一个对应的块第一次被装入高速缓存时, 它的有效位设成1。每个长字都有一个单独的脏位与之相联, 在写操作期间, 如果这个长字的数据发生改变, 那么它的脏位设成1。这个脏位保持这个设置, 直到这块的内容被写回到主存中。

当高速缓存被访问时, 地址的标志位与指定组的四位标志做比较。如果有一个标志与期望

块位置	每次循环后数据高速缓存中的内容				
	$j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

图5-21 相联映射数据高速缓存的内容

	每次循环后数据高速缓存中的内容					
	$j = 3$	$j = 7$	$j = 9$	$i = 4$	$i = 2$	$i = 0$
第0组	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)	A(0,0)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)	A(0,1)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)	A(0,3)
第1组						

图5-22 组相联映射数据高速缓存的内容

324

325

的地址匹配，并且相应块的有效位等于1，那么就命中了。图5-23给出了一个例子，在该例中被寻址的数据在第0组第4块的第3个长字中找到。

数据高速缓存可以在操作系统的控制下，使用直接写或写回协议中的任何一种。指令高速缓存的内容只有在读失效后装入新指令时才会改变。当一个新块必须装入一个已经满了的高速缓存组中时，替换算法随机选择被换出的块。当然，如果块中的一个或多个脏位等于1，那么必须首先执行写回操作。

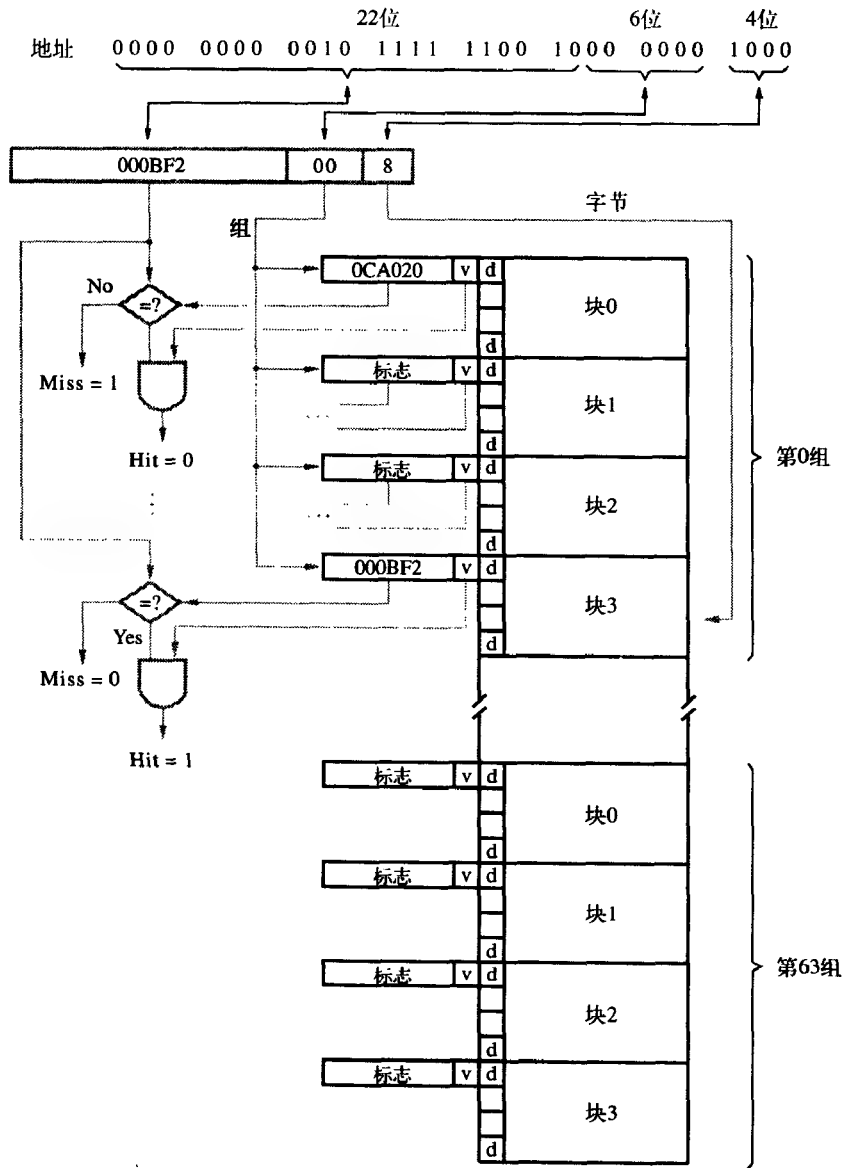


图5-23 68040微处理器的数据高速缓存结构

ARM710T高速缓存

ARM系列是由具有有效的RISC型体系结构，且具有低成本和低能耗特征的处理器组成。ARM710T是这个系列处理器中的一员。它只有一个高速缓存，同时用于指令和数据处理。



ARM710T高速缓存的组织结构与图5-23描述的高速缓存类似，它被排成4路组相联缓存。每个块包括16个字节，组成四个32位的字。

当处理器向高速缓存中写时，使用直接写协议。当新块需要空间时，使用随机替换算法决定哪一个高速缓存块被覆盖。

ARM710T高速缓存的结构与低成本、低能耗的目标一致。一个单独统一的高速缓存同时保存指令和数据，这比使用两个单独的高速缓存简单。直接写协议和随机替换算法也有助于实现的简单化。

### Pentium III高速缓存

Pentium III是高性能处理器。因为高性能依赖于高速访问指令和数据，所以Pentium III使用了二级高速缓存。第一级由一个16K字节的指令缓存和一个16K字节的数据缓存组成。数据缓存具有4路组相联结构，它既可以使用写回策略，也可以使用直接写策略。指令缓存具有2路组相联结构。由于正常情况下在程序执行期间指令不会被修改，所以指令高速缓存不需要写策略。

第二级高速缓存要大得多，它既保存指令，又保存数据。像图5-24中给出的那样，二级高速缓存连接到系统的其余部分。一个总线接口部件把高速缓存、主存和I/O设备连接起来。这里使用两个单独的总线：快速的高速缓存总线把二级高速缓存连接到处理器，而慢一些的系统总线用于连接主存和I/O设备。

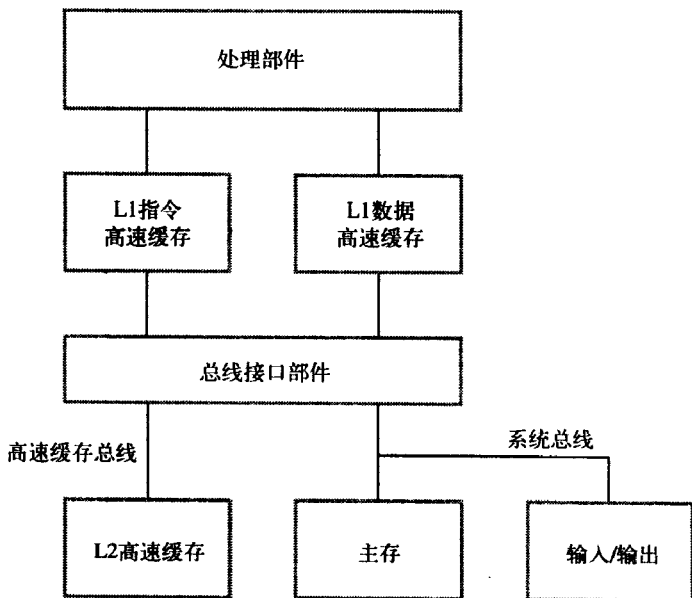


图5-24 Pentium III处理器的高速缓存和外部连接

二级高速缓存可以在处理器芯片外部实现，就像在Pentium III的一个称为Katmai的版本中所做的那样。在这种情况下，高速缓存包含512K字节，用静态随机存储器实现。它的结构是4路组相联结构，使用写回或直接写协议，可以在每块的基础上编程。高速缓存总线的宽度是64位。

VLSI技术的发展使得把二级高速缓存集成到处理器芯片的内部成为可能，这已经在Pentium III的Coppermine版本上实现了。这里高速缓存的容量是256K字节，它使用8路组相联结构。由于二级高速缓存在处理器芯片上，所以有可能使用更宽的256位高速缓存总线。

这些例子提出一个有趣的问题——二级高速缓存是在外部实现好还是在处理器芯片内部实

现好？外部实现的高速缓存允许更大的容量，但是它不便于加宽到处理器的数据连接通路，因为这需要额外的引脚，并且会增加输出驱动器的功耗。此外，外部高速缓存的时钟速度比较慢。Katmai的二级高速缓存按处理器时钟速度的一半驱动，而Coppermine的二级高速缓存以处理器时钟的全速驱动。把二级高速缓存放到处理器芯片上减少了访问的延迟，增加了带宽。因为数据可以使用更宽的通路传输，这可以带来更好的性能。集成式二级高速缓存的主要缺点是处理器芯片变得更大，使得它更加难以制造。

328

### Pentium 4高速缓存

Pentium 4处理器可以有高达三级的高速缓存。一级高速缓存包括分开的数据缓存和指令缓存。数据缓存容量为8K字节，按4路组相联方法组织。每个缓存块有64个字节。向这个缓存中写数据时使用直接写策略。可以在两个时钟周期内从数据缓存中访问整数数据。Pentium 4可以使用超过1.3GHz的时钟信号，这意味着数据可以在少于2ns的时间内被访问。指令缓存并不保存一般的机器指令，而是保存已经译码后的指令，这将在第11章中讨论。

二级高速缓存是一个统一的高速缓存，有256K字节的容量，按8路组相联方法组织。它的每个块包含128个字节。当向高速缓存中写时，使用写回策略。这个高速缓存的访问延迟是7个时钟周期。

一级高速缓存和二级高速缓存都在处理器芯片上实现，这个体系结构还允许包含一个芯片内的三级高速缓存。但是，在用于桌面计算机的Pentium 4处理器上没有实现这个高速缓存，这个高速缓存准备用在服务器系统的处理器芯片上。

## 5.6 性能因素

计算机在商业上取得成功的两个关键因素是性能和成本，其目标是以最低的成本获得最高的性能。在设计可用方案时的挑战是在不增加成本的前提下提高性能。衡量成效的一个通用标准是性价比。在这一节，我们讨论一些存储器设计的特征，它们能得到很好的性能。

性能依赖于机器指令能以多快的速度送入处理器，以及它们能以多快的速度执行。我们将在第7章和第8章讨论执行速度，展示如何使用附加电路在执行阶段提高指令处理的速度。在这一章，我们把重点放在存储器子系统上。

在5.4节描述的存储器层次结构源自于对更好性价比的探索，这个层次结构的主要目的是创建一个从处理器角度来看有短访问时间和大容量的存储器。层次结构的每一层都扮演一个重要的角色，在不同层之间的数据传输速度和效率也是非常重要的。在向或从快速部件中传输数据时，能以与该快速部件同等的速率进行是很有用的。如果慢速部件和快速部件使用同样的方式访问，实现相同访问速度是不可能的，但是如果慢速部件的组织结构中使用了并行机制的话，我们就可以实现这一点。引入并行机制的有效办法是使用交叉结构。

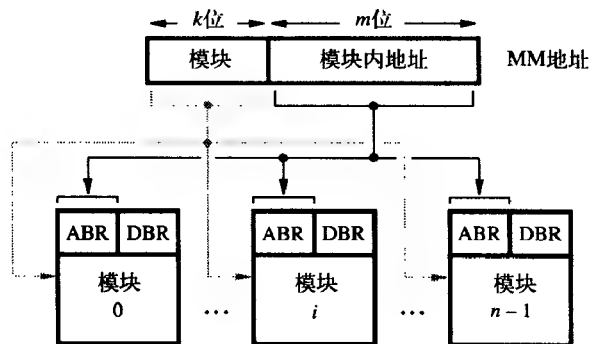
329

### 5.6.1 交叉

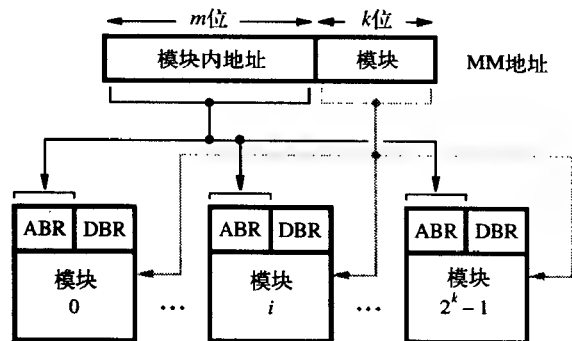
如果计算机的主存构造成为物理上分开的模块，每个模块都有它自己的地址缓冲寄存器（ABR）和数据缓冲寄存器（DBR），那么存储器访问操作可以同时多个模块中进行。这样，从或向主存系统传输多个字的总体速率就会提高。

在确定计算处理过程中能保持在工作状态的模块平均数时，如何把单独的地址分布在模块上是关键。图5-25给出了两种地址布局的方法。在第一种情况中，由处理器产生的地址按图5-25a

所示的方式进行译码。高  $k$  位指定  $n$  个模块中的一个模块，低  $m$  位指定在这个模块中的一个字节。在访问连续单元时，例如把一块数据传输到高速缓存中时，就只需要访问一个模块。而与此同时，有直接存储器访问（DMA）能力的设备可以访问另一个模块中的信息。



a) 连续字在一个模块中



b) 连续字在连续模块中

图5-25 在多模块存储器系统中寻址

第二个也是效率更高的模块编址方法在图5-25b中给出，它称为存储器交叉。存储器地址的低  $k$  位选择一个模块，而高  $m$  位指定在该模块中的一个单元。在这种方法中，连续的地址被放在相继的模块中。这样，生成访问连续内存单元请求的任何系统部件都能使多个模块同时处于工作状态，它可以在总体上加快块数据的访问，提高存储器系统的平均利用率。为了实现交叉结构，必须有  $2^k$  个模块，否则在地址空间中会有不存在的单元，出现空隙。

**例5.1** 交叉具有非常好的效果。考虑在读失效发生时从主存储器向高速缓存传输一块数据需要的时间。假设使用每块8个字的高速缓存，就像我们在5.5节中给出的举例那样。读失效时，包含需要字的块必须从主存中拷贝到高速缓存中。假设硬件有以下属性，它把地址送到主存储器要用一个时钟周期，主存储器使用相对较慢的DRAM芯片构建，允许在8个时钟周期内访问第一个字，但是块中后续字的每个字花费4个周期（回想5.2.3节，在DRAM中，当从一个给定行读取连续的单元时，行地址只需译码一次。然后使用阵列的连续列地址去访问需要的字，这时每次访问只需一半的时间）。此外，送一个字到高速缓存需要1个时钟周期。

在使用单一存储器模块时，将所需要的块装载到高速缓存中需要的时间是

$$1 + 8 + (7 \times 4) + 1 = 38 \text{ 周期}$$

现在假设存储器由四个交叉模块构成, 使用图5-25b中描述的方案。当块的起始地址到达存储器时, 所有的四个模块都开始使用高位地址来访问需要的数据。8个时钟周期后, 每个模块把一个字的数据放到它的DBR中。在接下来的4个时钟周期中, 这些数据被传输到高速缓存中, 每次一个字。在这段时间内, 每个模块的下一个字被访问, 然后它再花费4个时钟周期把这些字传输到高速缓存中。因此, 从交叉存储器中装入块需要的时间是

$$1 + 8 + 4 + 4 = 17 \text{ 周期}$$

因而, 交叉使得块传输时间减少了一半。

331

在5.2.4节中, 我们提到把交叉用于SDRAM芯片来提高访问连续字的速度。在大多数SDRAM芯片中存储阵列由两个或四个更小的交叉阵列存储器构成, 这样提高了向或从主存中传输一块数据时的速率。

### 5.6.2 命中率和失效开销

衡量存储器体系结构的具体实现效率的一个很好指标是在这个层次结构的各个层上访问信息的成功率。前面提到在高速缓存中一次成功的数据访问称为命中。命中次数比上访问总次数称为命中率, 而失效次数比上访问总次数称为失效率。

理想情况下, 整个存储器层次结构对于处理器来说表现得就像一个单独的存储器部件那样, 具有处理器芯片上的高速缓存的访问速度, 还具有磁盘的容量。我们距离这个理想目标有多远, 很大程度上取决于层次结构上不同层的命中率。超过0.9的高命中率对于高性能计算机来说是必需的。

失效发生后采取的必要措施会对性能产生影响。把所需信息放入高速缓存所需要的额外时间称为失效开销, 这个开销最终反映在处理器等待的时间上, 因为执行所需要的指令或数据不能获得。一般说来, 失效开销是把数据块从层次结构中的慢速部件传输到快速部件所需的时间。如果实现了在层次结构中不同部件之间传输数据的有效机制的话, 失效开销会降低。前面一节描述了交叉存储器是如何有效地降低失效开销问题的。

**例5.2** 现在考虑高速缓存对计算机总体性能的影响。令  $h$  表示命中率,  $M$  表示失效开销, 即访问主存信息需要的时间,  $C$  表示访问高速缓存信息需要的时间。处理器的平均访问时间是

$$t_{ave} = hC + (1 - h)M$$

我们使用例5.1中使用的参数。如果计算机没有高速缓存, 那么使用快速处理器和典型DRAM主存时, 每个存储器读访问花费10个时钟周期。假设计算机有一个每块8个字的高速缓存, 并且有一个交叉主存, 那么就像在5.6.1节中所示的那样, 装载一个块到主存需要17个时钟周期。假设在具有代表性的程序中30%的指令需要执行读写操作, 这意味着每执行100条指令会有130次存储器访问。假设高速缓存命中率对于指令是0.95, 对于数据是0.9, 我们再进一步假设读写操作的失效开销是相等的, 那么使用高速缓存带来的性能提高可以按下面公式粗略估算:

$$\frac{\text{无高速缓存时的时间}}{\text{有高速缓存时的时间}} = \frac{130 \times 10}{100 (0.95 \times 1 + 0.05 \times 17) + 30 (0.9 \times 1 + 0.1 \times 17)} = 5.04$$

这个结果表明使用高速缓存的计算机性能要比原来好五倍。

332

考虑这个高速缓存与命中率为100%的理想高速缓存（此时所有存储器使用都只耗费1个时钟周期）的相比效果也是很有趣的。我们对这些高速缓存相对性能的粗略估算如下

$$\frac{100 (0.95 \times 1 + 0.05 \times 17) + 30 (0.9 \times 1 + 0.1 \times 17)}{130} = 1.98$$

这表明处理器在高速缓存实际提供的环境中有效地工作在基于DRAM的主存中，而主存表现出来的速度只比高速缓存电路慢两倍。

在这个例子中我们做了一个简化的假设，那就是访问芯片内的高速缓存与通过系统总线访问主存使用相同的时钟周期。一个高性能处理器在比系统总线时钟快得多的时钟控制下运行，可能会快十倍。让我们来考虑在这种类型系统中高速缓存的影响。

**例5.3** 假设在处理器芯片上实现了一个单独的高速缓存，主存用SDRAM芯片实现。假设系统总线时钟比处理器时钟慢四倍。与例5.2一样，我们也假设高速缓存每块有8个字，并且高速缓存命中率对于指令是0.95，对于数据是0.9。SDRAM的时序图与图5-9相似。惟一的区别是使用8个字的脉冲操作，而不是4个字。于是根据图5-9，从RAS信号声明在主存和高速缓存之间传输一块数据开始到结束要耗费14个时钟周期。由于RAS和CAS信号就像图5-11所示的那样由存储器控制器产生，所以处理器把数据块的第一个字的地址送到存储器控制器中还需要1个时钟周期。因此传输一个块总共需要15个时钟周期。图5-9中所示的周期是系统总线时钟周期。如果处理器时钟要快四倍，那么需要60个处理器时钟周期来向或从主存中传输一个8个字的块。还要注意图5-9表明处理器可以在9个总线时钟周期内在主存中读写一个字，这包括图5-9所示的8个周期再加上向存储器发送地址需要的1个周期。因此，访问主存中的一个字需要36个处理器时钟周期。然而，处理器可以在1个处理器周期内访问高速缓存中的一个字。

重复例5.2中的计算可以得出：

$$\frac{\text{无高速缓存时的时间}}{\text{有高速缓存时的时间}} = \frac{130 \times 36}{100 (0.95 \times 1 + 0.05 \times 60) + 30 (0.9 \times 1 + 0.1 \times 60)} = 7.77$$

因而，考虑处理器与系统总线时钟速度的区别后表明高速缓存对性能有更大的正面作用。

前面的例子中，在考虑命中率时我们把指令和数据区分开来。虽然它们的命中率都超过0.9，但是指令命中率一般比数据命中率要高。命中率依赖于高速缓存的设计和所执行程序的指令和数据的访问模式。

如何才能提高命中率呢？一个明显的可能性是使用更大的高速缓存，但是这必定要增加成本。另一个可能性是在保持高速缓存总容量不变的前提下增加块的大小，以发挥局部空间的优势。如果在一个较大的块中所有的项数都是计算中所需的，那么最好是在一次失效后将所有这些项都装入高速缓存，而不是在多次失效后分别装入，每次只装入一个较小的块。对交叉主存块的并行访问效率是获得这个优势的基本前提。在达到一定大小之前，增大块的大小是有效的，但是对命中率的进一步提高最终会被抵消掉，因为在更大的块中，有些项还是没有被访问，该块就已经被换出了。随着块大小的增加，失效开销也会加大。由于计算机性能受命中率增加的正面影响，还受失效开销增加的负面影响，因此块大小适中时可以获得最佳效果。在实际中，最常用的块大小范围是16字节到128字节。

最后，我们注意到如果在新块装入高速缓存时使用直接装入法，那么失效开销可以降低。

这样,处理器在所需要的字一旦被装入高速缓存后就可以继续工作,而不是必须等待传输完成。

### 5.6.3 处理器芯片上的高速缓存

当信息在不同的芯片间传输时,芯片上的驱动器和接收器会产生不可忽略的延迟。所以从速度的角度考虑,放置高速缓存的最佳位置是在处理器芯片上。但不幸的是,处理器芯片上的空间还需要用于很多其他的功能,这限制了能放下的高速缓存的容量。

所有高性能处理器芯片都包含某种形式的高速缓存。一些制造商选择实现两个独立的高速缓存,一个用于指令,另一个用于数据,如68040、Pentium III和Pentium 4处理器。其他的只实现一个单一的高速缓存,既用于指令又用于数据,如ARM710T处理器。

334

指令与数据混合的高速缓存在一定程度上可能有更高的命中率,因为在把新信息映射到高速缓存中时它提供了更大的灵活性。但是,如果使用分开的高速缓存,就可以同时访问两个高速缓存,这增加了并发性,带来更好的性能。分离高速缓存的缺点是为了增加并发性需要更复杂的电路。

在高性能处理器中通常使用二级高速缓存。一级高速缓存在处理器芯片上,而二级高速缓存要大得多,可以用SRAM芯片在外部实现。但是也可以在处理器芯片上实现一个稍微小一些的二级高速缓存,如同在5.5.4节中描述的Coppermine版本的Pentium III处理器那样。

如果同时使用了一级高速缓存和二级高速缓存,那么应该把一级高速缓存设计成允许处理器使用非常快的速度访问,因为它的访问速度对处理器的时钟速率有很大影响。访问高速缓存的速度不可能与访问寄存器相同,因为高速缓存要大得多,因而它更复杂。加快访问高速缓存速度的一个实用方法是同时访问多个字,然后让处理器使用这些字,每次一个。这项技术在很多商业处理器上被应用。

二级高速缓存可以慢一些,但是它应该比一级高速缓存大得多,以保证高命中率。速度不是那么关键,因为这只影响一级高速缓存的失效开销。用作工作站的计算机可能有一个容量为数十K字节的一级高速缓存和数兆字节的二级高速缓存。

二级高速缓存进一步减小了主存速度对计算机性能的影响。在有二级高速缓存的系统中,处理器平均访问时间是

$$t_{ave} = h_1 C_1 + (1 - h_1) h_2 C_2 + (1 - h_1) (1 - h_2) M$$

其中,

$h_1$ 是一级高速缓存的命中率;

$h_2$ 是二级高速缓存的命中率;

$C_1$ 是一级高速缓存的信息访问时间;

$C_2$ 是二级高速缓存的信息访问时间;

$M$ 是主存的信息访问时间。

二级高速缓存的失效次数由  $(1 - h_1) (1 - h_2)$  项给出,它应该很低。如果  $h_1$  和  $h_2$  都是在高于90%这个范围内,那么失效次数将小于处理器内存访问次数的1%。那么,从性能的角度看,失效开销  $M$  的重要性就小一些。参见习题5.18对这个问题的定量分析。

### 5.6.4 其他改进

除了刚才讨论的主要设计问题,还存在其他几种改进性能的可能性。在这一节我们讨论其

中三种。

### 写缓冲区

当使用直接写协议时，每个写操作导致一个新数据被写入主存。如果处理器必须等待存储器功能完成，像我们之前假设的那样，那么所有的写请求导致处理器变慢。但是处理器通常并不直接依赖于写操作的结果，所以它不必等待写请求的完成。为了提高性能，可以引入一个写缓冲区来临时保存写请求。处理器把每个写请求放到这个缓冲区中，然后继续执行后面的指令。当存储器没有可响应的读请求时，就把保存在缓冲区中的写请求发给主存。注意，立即满足读请求是很重要的，因为如果没有从存储器中读取的数据，处理器通常就不能继续执行。因此，这些请求的优先级比写请求高。

写缓冲区可以保存一定数量的写请求，因此，后面的读请求引用的数据可能仍在写缓冲区中。为了保证正确的操作，从存储器中读取的数据地址要与写缓冲区中数据的地址做比较，如果匹配，那么写缓冲区中的数据就是可用的。

如果使用写回协议，那么就会有不同的情形。这时，写操作只是简单地对高速缓存中相应的字进行写。读失效导致新数据块被放入高速缓存，这会替换出一个已存在的包含一些脏数据的块。考虑一下这时会发生什么，脏块必须被写到主存中。如果先执行所需的写回操作，那么处理器必须等待很长时间，等待把新块读入高速缓存。因此首先读入新块是更合理的。可以设计成在读取新块时提供一个快速的写缓冲区，用来临时存储从高速缓存中换出的脏块，然后把缓冲区中的内容写进主存。因而，写缓冲区对于写回协议也是有效的。

### 预取

在前面讨论的高速缓存机制中，我们假设新数据在第一次需要时被装入高速缓存。一个读失效发生，然后需要的数据从主存装入，处理器必须暂停，直到新数据到达，这就是失效开销的影响。

为了避免处理器停止，可以在需要数据之前把它们预取到高速缓存中。实现这一点的最简单方法是使用软件。处理器的指令集可能提供一个特殊的预取指令。执行这个指令使得地址指向的数据被装入高速缓存，就像读失效时的情况一样。然而，处理器并不等待引用的数据。在程序中插入预取指令，使得数据在程序需要时已经被装入到了高速缓存中。最好在处理器执行不会引发读失效指令时开始预取，这样主存访问就能与处理器运算重叠。

预取指令可以由程序员或编译器插入。显然让编译器插入这些指令更好，对很多应用程序来说它可以做得很成功。注意软件预取肯定有一定的开销，因为包含预取指令会增加程序的长度。此外，一些预取操作可能会把那些后续指令不用的数据装入高速缓存。如果其他数据引发的读失效把已预取的数据从高速缓存中换出，这种情况就会发生。但是，软件预取对性能的总体效果还是有利的，并且已有支持这个特性的机器指令存在。关于软件预取的全面讨论见参考文献[1]。

预取也能通过硬件实现，这需要增加用于发现存储器引用模式并根据这个模式预取数据的电路。为了这个目标已经提出了很多方案，它们超出了本书的范围。对这些方案的描述可以在参考文献[2]和参考文献[3]中找到。

Intel的Pentium 4处理器具有把信息预取到高速缓存的装置，它同时使用了软件和硬件的方法。程序中可以包含特殊的预取指令，把数据块放入所需级别的高速缓存中。硬件控制的预取

功能根据以前的模式把需要的块放入二级高速缓存中。

### 无锁定高速缓存

如果上面讨论的软件预取方案对指令的正常执行产生很大干扰的话,就不会产生好的效果。如果预取操作在预取完成前阻止其他对高速缓存的访问时就会出现这样的情况。这种类型的高速缓存在它响应一次失效时,我们称它为被锁定了。可以通过修改高速缓存的基本结构来解决这个问题,允许处理器在高速缓存响应一次失效时仍能访问它。实际上,我们希望能支持多个未响应的失效。

能支持多个未响应失效的高速缓存称为无锁定高速缓存。因为一次只能响应一个失效,因此必须引入用于跟踪所有未响应失效的电路。这可以通过把与失效相关的信息保存在特定的寄存器中来实现。无锁定高速缓存在20世纪80年代早期最先用于由Control Data公司<sup>[4]</sup>生产的Cyber系列计算机上。

我们已经把软件预取看作是在读失效时不锁定高速缓存的主要动机。一个更重要的原因是使用流水线结构的处理器中,多个指令的执行是重叠的,一条指令引起的读失效会停止其他指令的执行。无锁定高速缓存减少了这种停止的可能性。我们在第8章中会再提到这个主题,在那里将详细分析流水线结构。

## 5.7 虚拟存储器

在大多数现代计算机系统中,物理主存空间没有处理器生成的地址所能跨越的空间大。例如,一个产生32位地址的处理器有4G的可寻址空间,而典型的计算机中主存容量的范围只是从几百兆到1G字节。在一个程序还没有完全装入主存中时,它当前未被执行的部分存放在辅助存储设备上,例如磁盘。当然,程序最终要执行的所有部分都要预先装入主存。当一个新的程序段要移入一个已经满了的主存时,必须替换已经在主存中存在的另一个程序段。在现代计算机中,操作系统自动在主存和辅助存储设备之间传送程序和数据。因此,应用程序员不需要知道可用主存所具有的限制。

在执行中需要程序块和数据块时自动把它们移入主存的技术称为虚拟存储器技术。程序,其实也就是处理器,引用了一个与可用物理主存空间无关的指令和数据空间。处理器生成的指令或数据的二进制地址称为虚拟或逻辑地址。这些地址用硬件和软件结合的方法转换成物理地址。如果一个虚拟地址指向程序空间或数据空间的一部分,而这部分当前位于物理主存中,那么主存中对应位置的内容可以立即被访问。而相反情况,如果引用地址不在主存中,那么在使用它之前必须把它的内容放入主存中恰当的位置上。

图5-26给出了实现虚拟存储器的典型组织结构。一个称为存储器管理部件(MMU)的特殊硬件把虚拟地址转换成物理地址。当需要的数据(或指令)在主存中时,这些数据通过前面描述的高速缓存机制来获取。如果数据不在主存中,那么MMU通知操作系统把数据从磁盘上装入主存。磁盘和主存

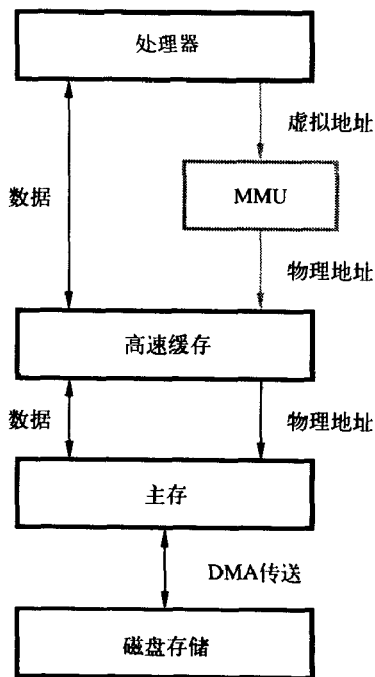


图5-26 虚拟存储器组织结构

[337]

[338]



之间的数据传送使用第4章中讨论的DMA方式进行。

## 地址转换

把虚拟地址转换成物理地址的简单方法就是假设所有的程序和数据由称为页的相同长度单元组成，每个页中包含主存占用连续单元的一块数据。页的长度通常是从2K到16K字节。当转换机制确定需要做一次移动时，页构成了在主存和磁盘之间移动信息的基本单位。页不能太小，因为对磁盘访问时间（几毫秒）比对主存访问时间要长得多。造成这个的原因是磁盘需要花费相当长的时间对数据做定位，但是一旦定位，数据就能以每秒几兆的速度传输。另一方面，如果页设置太大，那么页中很大一部分数据可能没有被用到，但是这些不需要的数据会占用主存的有用空间。

这个讨论与5.5节中在高速缓存上引入的概念类似。高速缓存在处理器与主存之间的速度沟壑上搭起一座桥梁，它使用硬件实现，而虚拟存储器机制是在主存和辅助存储设备的速度沟壑上架桥，它的实现通常部分地使用了软件技术。从概念上看，高速缓存技术和虚拟存储器技术非常相似，它们的区别主要在实现细节上。

虚拟存储器地址的转换方法是基于图5-27中所描述的固定长度页的概念实现的。处理器产生的每个虚拟地址，不管是用于取指令还是存取操作数，都被解释成虚拟页号（高地址位）后边跟一个偏移量（低地址位），用来在一个页内指定一个特定字节（或字）的位置。每个页在主存中的位置信息保存在页表中。这个信息中包括页所在的主存地址和当前的状态。主存中能保

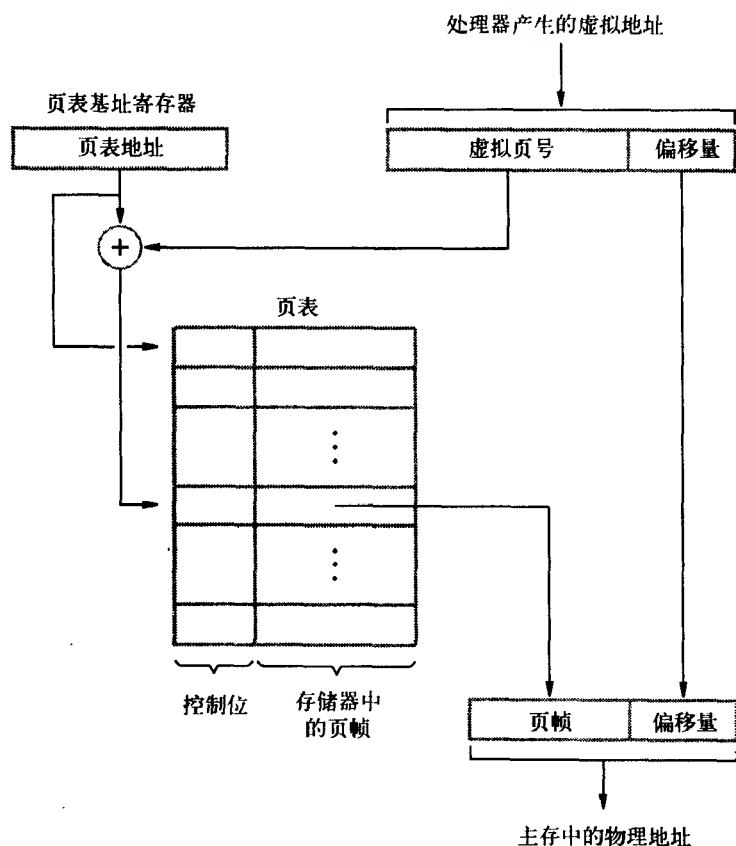


图5-27 虚拟存储器地址转换

存一个页的区域称为页帧。页表的开始地址保存在页表基址寄存器中。把虚拟页号与这个寄存器的内容相加就能得到这个页在页表中相应入口的地址。如果该页当前位于主存中，那么这个单元的内容指出这个页的起始地址。

页表中的每个入口还包含一些控制位，当该页在主存中时，控制位用来描述它的状态。其中一位用来表示页的有效性，即这个页是否确实在主存中。这个位可以允许操作系统标志一个页无效，而不用实际移除这个页。另一个位表示页在主存期间是否被修改过。与高速缓存一样，这个信息用于决定在为其他页腾出空间而移除这个页之前是否需要把它的内容写回到磁盘中。还有一个控制位表示对访问这个页的不同约束。例如，一个程序可能有完全的读写权限，它也可能被规定成只能读取。

[339]

MMU使用的页表信息用于所有的读写访问，所以从理想角度考虑，页表应该放在MMU内部。不幸的是，页表可能很大，由于MMU一般作为处理器芯片的一部分来实现（与主高速缓存一起），所以不可能把整个页表放到芯片内。因此将页表保存在主存中。但是，页表的一小部分的拷贝可以放入MMU，这个部分包括那些最近访问页所对应的入口。为了这个目的，把一小块通常称为转换监视缓冲区（TLB）的高速缓存并入MMU。对主存中页表的TLB操作本质上与在讨论高速缓存时提到的操作相同。除了组成页表入口的信息外，TLB中还必须包含入口的虚拟地址。图5-28显示了TLB可能的一种组织结构，这里使用了关联映射技术。商业产品中也有使用组相联映射技术的。

[340]

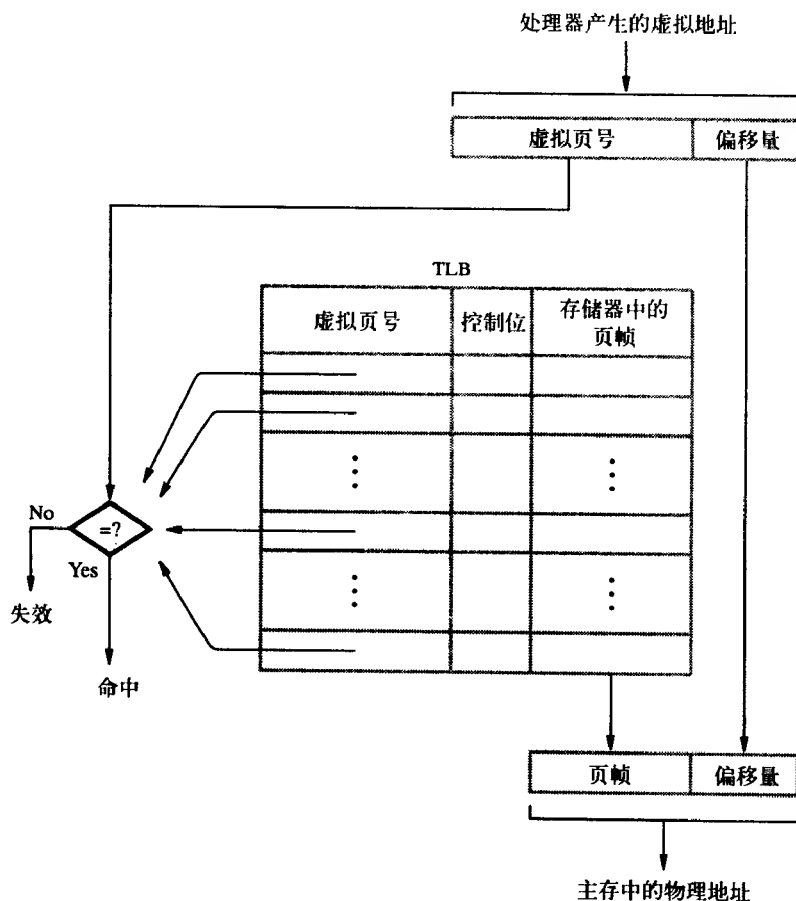


图5-28 相联映射TLB的使用

[341]

TLB中的内容与主存中页表的内容保持一致是一个基本的要求。当操作系统更改页表中的内容时,它必须同时把TLB中对应入口设成无效。TLB中的一个控制位用于这个目的。当一个入口无效时,TLB将获取新的信息,这是MMU对访问失效进行响应的一个步骤。

接下来是地址变换处理。给定一个虚拟地址,MMU在TLB中搜索引用的页。如果这个页的页表入口在TLB中,则可以直接获得该物理地址。如果TLB失效,那么就从主存中的页表获取需要的入口,TLB同时被更新。

当一个程序发出访问一个不在主存中的页请求时,我们说发生一次页故障。在能进行访问之前必须把整个页从磁盘放入主存中。当检测到一个页故障时,MMU通过产生一个异常(中断)来请求操作系统进行处理。这时,当前执行的任务被中断,控制权转移到操作系统。操作系统把请求的页从磁盘拷贝到主存中,然后把控制权交还给被中断的任务。因为页传输时会有个很长时间的延迟,所以操作系统可能挂起引起页故障的任务,然后开始执行另一个页在主存中的任务。

必须保证被中断的任务在恢复执行后能正确地继续执行。当某个指令访问不在主存中的存储器操作数时会发生一个页故障,它在这个指令完成前引发一个中断。因此,当这个任务恢复执行时,被中断的指令要么从中断点继续执行,要么重新开始。具体处理器的设计决定了具体使用的是哪一种方法。

当主存已经满的时候,如果从磁盘读出一个新的页,那么必须替换一个已经存在的页。选择哪个页被换出问题的重要性与在高速缓存中一样,程序把大多数时间花费在一个局部区域中的规律在这里也同样适用。因为主存要比高速缓存大得多,所以应该能把相对应的程序中更多的部分放入到主存中,这将会减少向或从磁盘中传输数据的频率。与LRU替换算法类似的概念也可以用于页替换中,页表入口中的控制位可以指明使用的情况。一个简单的方案是基于一个控制位的,当相应的页被引用(访问)时,这个控制位被设成1。操作系统会不时地清除页表所有入口的这个位,这样提供了一个简单的方法来判断最近哪一个页没有被使用过。

一个修改过的页在移出主存之前需要写回到磁盘中。注意用于高速缓存存储器结构中的直接写协议不适合虚拟存储器,这一点很重要。磁盘的访问时间太长,经常向里面写入少量数据是没有意义的。

MMU中的地址转换处理需要一些时间来执行,这主要依赖于搜索TLB的入口所需要的时间。由于引用的局部性,所以连续的地址转换很可能只涉及同一个页上的地址。这种情况在取指令时更明显。这样,我们可以引入一个或多个特殊寄存器来保存最近进行转换的虚拟页号和物理页帧,从而减少平均转换时间。访问这些寄存器中的信息可以比访问TLB更快。

## 5.8 存储器管理需求

在对虚拟存储器概念的讨论中,隐含地假设只有一个大程序在执行。如果不能把这个程序全部装入物理存储器,那么它的一部分(一些页)在执行时将从磁盘移到内存中。虽然我们间接地提到了管理程序段移动所需要的软件程序,但是并没有指出它们的细节内容。

管理程序是计算机操作系统的一部分。操作系统程序可以很便利地被组装到一个虚拟地址空间中,这个空间称为系统空间,它与用户程序所在的虚拟地址空间相分离,后一个空间称为用户空间。实际上,可能存在多个用户空间,每个用户一个,这可以通过为每个用户的程序提供一个单独的页表来实现。MMU使用页表基址寄存器来判断在转换过程中使用的页表,因此,通过修改这个寄存器的内容,操作系统就可以从一个空间转换到另一个空间。于是,物理主存储器由系统空间和

[342]

多个用户空间的活动页共同分享,但是,在给定时刻,只有属于其中一个空间的页能被访问。

在任何允许独立用户程序在内存中共存的计算机中,必须解决保护的问题,任何程序都不能破坏内存中其他程序的指令和数据。这种保护可以通过几种方法来提供。让我们首先来考虑最基本的保护形式。回想一下最简单的情况,处理器有两个状态,分别是管态和用户状态。就像名称所示的那样,当执行操作系统程序时,处理器处于管态,而执行用户程序时,处理器处于用户状态。在用户状态下,有一些指令处理器不能执行。这些特权指令,包括像修改页表基址寄存器这样的操作,只能在处理器处于管态时才能执行。因此,用户程序不能访问其他用户空间的页表和系统空间的页表。

有时候一个应用程序要求访问属于另一个程序的页,操作系统通过使这个页在两个空间中都可见来实现这一点,因而这个共享的页在两个不同的页表中都有入口。在每个页表中入口的控制位可以用来控制相应程序的访问权限。例如对一个给定的页,一个程序可能允许读和写,而另一个程序只有读权限。

343

## 5.9 辅助存储器

前面章节讨论的半导体存储器并不能提供计算机需要全部存储能力,主要限制是每位存储信息的成本。大多数计算机系统的大量存储需求使用更经济的磁盘、光盘和磁带实现,它们通常称为辅助存储器设备。

### 5.9.1 磁性硬盘

顾名思义,在磁盘系统中存储介质由安装在一个轴上的一个或多个磁盘组成。在每个盘上覆盖着一层很薄的磁性薄膜,通常两面都有。盘放在一个旋转的驱动器上,这样磁性表面就可以在读写磁头附近移动,像图5-29a所示的那样。各个磁盘以一个统一的速度旋转。每个磁头包括一个磁轭和一个磁性线圈,如图5-29b所示。

数字信息可以通过向磁性线圈施加适当极性的电流脉冲来存储到磁性薄膜上,这导致磁头下的薄膜区域的磁化方向与施加的磁场同向。这个磁头还可以用于读取存储的信息,此时,磁性薄膜相对磁轭的运动导致磁头附近的磁场发生变化,这会在磁性线圈中感应出电压,这时这个线圈用作传感线圈。由控制电路检测到的电压正负用来判断薄膜的磁化状态。只有在读操作时才能感应到磁头下磁场的变化。因此,如果二进制状态0和1用磁化的两个相反的状态表示,那么只有在位流中0变成1或1变成0时,磁头才能感应出电压,一长串0或1只有在这个串的开头和结尾才能产生感应电压。为了判断存储的连续0或1的个数,必须用一个时钟提供同步信息。在早期的设计中,时钟被存储在一个单独的磁道中,这个磁道在每个位周期中磁性都必须改变。以这个时钟信号为参照,存储在其他磁道中的数据就能正确读出了。

现代的方法是把时钟信息结合到数据中,现在已经开发出几种不同的技术用于这种编码。一种简单的方案在图5-29c中给出,它被称为相位编码或曼彻斯特编码。在这种方案中,每个位都有磁性变化发生,如图所示。注意要确保磁性变化在每个位周期的中点发生,以此来提供时钟周期。曼彻斯特编码的缺点是它的位存储密度较低,表示一位需要的空间必须足够大,以容纳磁性的两个变化。曼彻斯特编码的例子阐述了如何实现一个自同步时钟方案,因为它比较好理解。人们还开发了一些更紧凑的编码,它们更有效,能提供更高的存储密度,但同时也需要更复杂的控制电路。对这些编码的讨论超出了本书的范围。

344

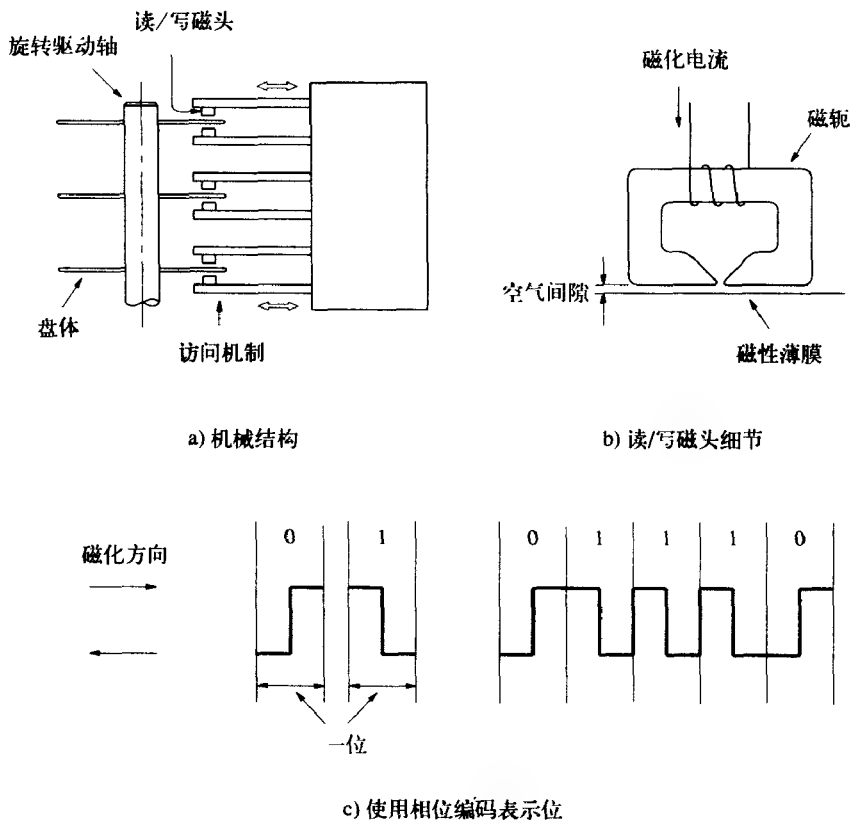


图5-29 磁盘原理

读写磁头必须与转动的盘面保持一个很近的距离，以此来获得较高的位密度和可靠的读写操作。当磁盘以一个稳定的速度旋转时，在盘面和磁头之间会产生一个空气压力，迫使磁头远离盘面。这个力可以由一个弹簧装置抵消，这个装置把磁头压向盘面。在磁头和它的支撑臂之间有灵活的弹簧连接，这允许磁头在离开盘面所需距离的位置上悬置，可以免受磁盘表面微小起伏的影响。

在多数现代磁盘部件中，盘体和磁头被放在一个密封的、对空气进行过滤的外壳中，这个方法称为温切斯特技术。在这样的部件中，读写磁头可以在距离磁化轨迹表面更近的位置上工作，因为里面不存在灰尘微粒，而灰尘微粒是未密封组装中的难题。磁头离磁道表面越近，数据就能以越高的密度沿着磁道存放，磁道之间的距离也越近。因此，在给定物理尺寸的条件下，温切斯特磁盘的容量比未密封部件更大。温切斯特技术的另一个优势是在密封部件中，存储介质没有暴露在污染环境中，因而数据有更好的完整性。

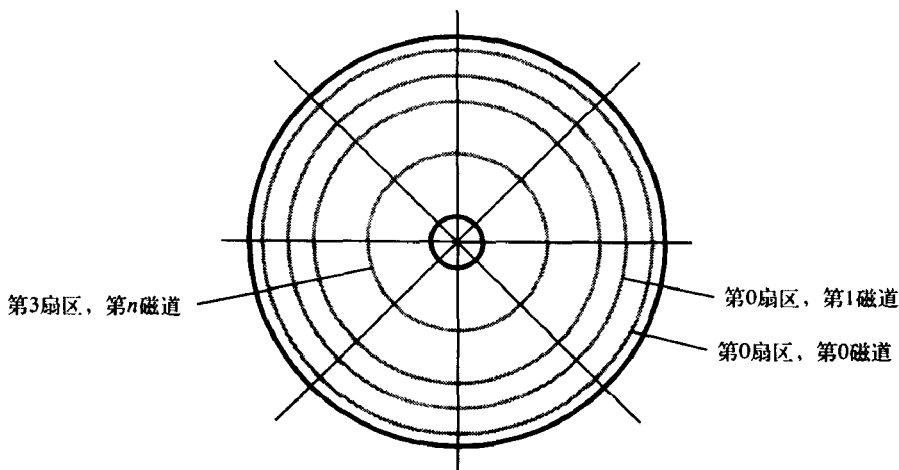
磁盘系统中的读/写磁头是可移动的。每个盘面有一个磁头，所有的磁头都安装在一个梳状支撑臂上，这个支撑臂可以在半径方向上横跨磁盘的磁道运动，实现了对单个磁头的访问，如图5-29a中所示。要在一个给定的磁道上读或写数据，装有读写磁头的支撑臂必须首先定位到该磁道上。

磁盘系统包括三个关键部分。第一个部分是装配的盘体，通常称为磁盘。第二个部分是电机装置，用来旋转磁盘和移动读写磁头，这称为磁盘驱动器。第三个部分是控制系统运作的控制电路，称为磁盘控制器。磁盘控制器可以实现成一个单独的模块，也可以放在包含整个磁盘

系统的外壳中。我们已经注意到通常把磁盘驱动器与磁盘合起来称为磁盘，在后面的章节中如果没有歧义，我们也使用这种叫法。

### 磁盘数据的组织结构和访问

磁盘上数据的结构如图5-30所示，每个面都被分成同心的磁道，每个磁道又分成扇区。各个表面上相同磁道的集合形成逻辑上的柱面，不需移动读写磁头就可以访问一个柱面中各个磁道上的数据。数据通过指定表面号、磁道号和扇区号来访问，读写操作在扇区边界处开始。



346

图5-30 磁盘一个表面的结构

数据位顺序存在每个磁道中。每个扇区通常包含512字节的数据，但是也有使用其他尺寸的。数据的前面有一个扇区头，包含标志（寻址）信息，这些信息用来在选定的磁道上找到需要的扇区。在数据后面有一些由纠错码（ECC）组成的附加位。ECC位用来检测和纠正在读写这512字节数据时可能发生的错误。为了便于区别两个连续的区，扇区间存在一个扇区间隙。

一个未格式化磁盘的磁道上没有任何信息，格式化过程把磁盘从物理上分成磁道和扇区。这个过程可能会发现一些有缺陷的扇区，甚至是整个磁道，磁盘控制器保存这些缺陷的记录，并在使用时排除这些扇区。格式化后的磁盘容量是一个给定磁盘存储容量的很好的指标。格式信息占磁盘能存储的全部信息的15%，它包括扇区头、ECC位和扇区间隙。在一台典型的计算机中，磁盘又被分成逻辑分区。至少存在一个这样的分区，称为主分区，可以有多个附加的分区。

图5-30说明每个磁道有相同数目的扇区，所以所有的磁道有相同的存储容量。因此，存储的信息在靠里的磁道上比靠外的磁道放得更密。这种布局在很多磁盘上使用，因为它简化了访问数据需要的电路。但是，我们可以通过在周长更长的外侧磁道上放置更多的信息来增加存储密度，这需要使用更加复杂的访问电路。这种方案用在大型磁盘上。

### 访问时间

从接收地址到开始实际传输数据之间的时间延迟包含两个部分。第一个部分称为寻道时间，是把读写磁头移动到恰当磁道所需的时间，它依赖于开始时磁头到地址对应磁道的相对位置。寻道时间的平均值在5~8毫秒的范围内。第二个部分称为旋转延迟或等待时间，它是从磁头定位到正确磁道直到地址指定扇区的起始位置经过磁头下方所用的时间。平均起来，这是磁盘转动半周所花的时间。这两个延迟的总和称为访问时间。如果在一次操作中只有少数扇区的数据被传输，那么访问时间至少比实际数据传输周期高一个数量级。

### 典型的磁盘

现在使用的3.5英寸（直径）高密度高数据速率磁盘有以下典型参数。它有20个数据记录面，每个面有15 000个磁道，每个磁道有400个扇区，而每个扇区包含512个字节的数据。于是，格式化过的磁盘总容量是 $20 \times 15\,000 \times 400 \times 512 \approx 60 \times 10^9 = 60\text{G}$ 。平均寻道时间是6毫秒。盘体转速是每分钟10 000转，所以平均等待时间是3毫秒，即转半圈的时间。从磁道到磁盘控制器的数据缓冲区的平均内部传输速率是每秒34M字节。当连接到SCSI总线时，这种类型的驱动器可以有每秒160M的外部传输速率。因此需要一个缓冲方案来处理传输速度上的不同，这在下一节解释。

[347]

也有一些很小的磁盘。例如一英寸的磁盘可以存储1G的数据，它的物理尺寸与一个纸板火柴相当，重量小于1盎司。这样的磁盘在便携设备和手持设备应用中很有吸引力。在一个数码相机中，一个这样的磁盘可以存储1000幅照片。现在看IBM公司在1980年生产的第一个容量为1G的磁盘很有意思，它有一个厨具那么大，重达250千克，价值40 000美元。

### 数据缓冲区/高速缓存

一个磁盘驱动器使用一些标准的连接方案连到计算机系统的其他部分上，通常使用标准总线，例如在4.7.2节中讨论的SCSI总线。包含所需的SCSI接口电路的磁盘驱动器通常称为SCSI驱动器。SCSI总线传输数据的速率要比从磁盘的磁道读数据的速率快得多，处理磁盘和SCSI总线之间的传输速率差异的一个有效方法是在磁盘部件中引入一个数据缓冲区。这个缓冲区是一个半导体存储器，能存储几兆的数据。请求的数据在磁盘磁道和缓冲区之间传输，传输速率依赖于磁盘转速。在数据缓冲区和总线上其他设备之间的传输能以总线允许的最大速度进行，这里所说的其他设备通常是指主存。

数据缓冲区还能用来为磁盘提供高速缓存机制。当一个读请求到达磁盘时，控制器可以先检查所要的数据是否已经在高速缓存（缓冲区）中。如果在，那么可以在微秒级访问数据并把它们放在SCSI总线上，而不是毫秒级。如果不在，那么使用通常的办法把数据从磁盘磁道上读出，然后存入高速缓存中。由于后面的读请求很可能会请求当前访问数据后续的数据，所以磁盘控制器可以读取比需要数据更多的数据，并把它们放入高速缓存，这样会潜在地缩短下一次请求的响应时间。这个高速缓存通常足够大，能放下整个磁道的数据，所以一个可能的策略是读写磁头一到达所需磁道上方就开始传输这个磁道的内容。

### 磁盘控制器

磁盘驱动器的操作由磁盘控制器电路控制，此外这个电路还提供磁盘驱动器和总线之间的接口，总线用来把磁盘驱动器连到计算机系统的其他部分。磁盘控制器可以用来控制多个驱动器。图5-31显示了一个磁盘控制器，它控制了两个磁盘驱动器。

[348]

磁盘控制器直接连到处理器的系统总线或扩展总线上，其中包括可由操作系统读写的多个寄存器。于是操作系统和磁盘控制器之间的通讯可以使用与I/O接口相同的方法实现，就像在第4章中讨论的那样。磁盘控制器使用DMA方案在磁盘和主存之间传输数据。实际上，这些传输是从或者向数据缓冲区进行的操作，它

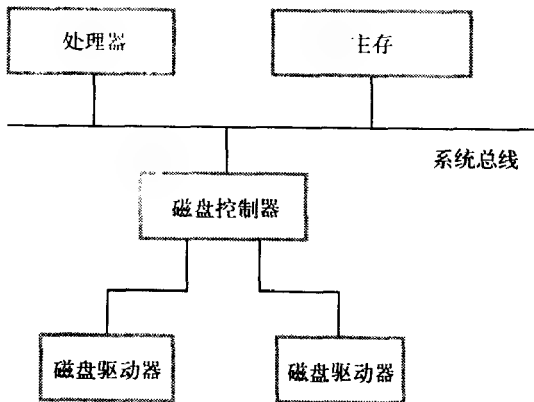


图5-31 连在系统总线上的磁盘

作为磁盘控制器的一部分来实现。操作系统发出读或写请求启动一次传输，这需把所需的地址和控制信息装入控制器寄存器，通常是：

主存地址——传输中涉及的字块的第一个主存单元的地址。

磁盘地址——包含所需字块起始位置的扇区单元。

字数——传输块中的字数。

操作系统发出的磁盘地址是逻辑地址，它与磁盘上对应的物理地址可能不一样。例如，磁盘在格式化时检测到了坏扇区，磁盘控制器记录这些扇区，然后用其他扇区代替。通常，在每个磁道或同一个柱面的其他磁道中保留了一些空闲扇区，用作坏扇区的替代品。

从磁盘驱动器的角度来看，控制器的主要功能是：

寻道——使磁盘驱动器把读写磁头从当前位置移动到需要的磁道上。

读——初始化一个读操作，从磁盘地址寄存器指定的地址开始。从磁盘顺序读出的数据被组装成字，并被放入数据缓冲区，用来向主存传输。字数由字计数寄存器决定。

写——使用类似于读操作的控制方法把数据传输到磁盘上。

错误检查——计算从指定扇区读出的数据的纠错码（ECC）值，把它与从磁盘读出的相应纠错码值做比较。不匹配时，如果能纠正，它就纠正这个错误，否则产生一个中断，通知操作系统发生了一个错误。在写操作期间，控制器计算要写入的数据的ECC值，并把这个值存到磁盘上。

349

如果磁盘驱动器连接到一个分包传输的总线上，那么控制器必须能够处理这样的传输。例如，一个用于SCSI驱动器的控制器要遵从第4章描述的SCSI协议。

### 软件和操作系统的含义

所有涉及磁盘的数据传输活动都是由操作系统发起的。磁盘是非易失存储介质，所以操作系统本身也存储在磁盘上。在计算机正常操作期间，操作系统的一部分被装入主存，按需要进行执行。

当电源关闭时，主存的内容都丢失了。当电源重新打开时，操作系统必须重新装入主存，这是引导程序的一部分工作。为了开始引导，主存的一小部分用非易失的ROM实现。在ROM中存储一个小的监控程序，能读写主存，也能读取存储在磁盘地址0上的一块数据。这个块称为引导块，它包含有一个装载程序。当ROM监控程序把引导块装入主存后，引导块把操作系统的其他部分导入主存。

与主存访问相比，磁盘访问非常慢，这主要是由于长时间的寻道。当操作系统初始化一个磁盘传输操作后，它通常转向另一个任务的执行，以保证时间的有效利用，否则这个时间被用来等待传输的完成。磁盘控制器通过产生一个中断来通知操作系统传输完成。

在包含多个磁盘的计算机系统中操作系统可以涉及多个需要的磁盘传输。如果一个磁盘正在进行从它或向它的DMA传输时，另一个磁盘在做寻道，那么就可以获得高效的操作。操作系统可以安排这些重叠的I/O活动。

### 软盘

前面讨论的设备叫做硬盘部件，软盘是更小、更简单、更廉价的磁盘部件，它包含一个柔软的可移动的塑料磁盘，这个磁盘覆盖着一层磁性材料。磁盘装在一个塑料的外壳中，这个外壳带一个开口，读写磁头从开口处接触磁盘。磁盘驱动器的旋转轴可以插入磁盘中心的一个孔中转动磁盘。

软盘在存储数据时使用的一个最简单方案是前面提到的相位或曼彻斯特编码。我们说按这



种方法编码的磁盘只有单密度。这个方案的一个更复杂的变体称为双密度，它通常用于现在的标准软盘。双密度方案把存储密度增加了一倍，但是同时也需要在磁盘控制器中使用更复杂的电路。

软盘的主要特点是低成本和可以方便地移动，但是与硬盘相比，它们的容量小得多，访问时间更长，并且失败率更高。当前标准的软盘直径为3.25英寸，存储1.44M或2M字节的数据。现在也有更大的超级软盘，这样的软盘称为zip盘，能存储超过100M字节的数据。近年来，随着可擦写光盘的出现，软盘技术的吸引力正在下降。光盘在下面讨论。

### RAID磁盘阵列

处理器速度在过去的10年中显著地增加，处理器性能每18个月就翻一番。半导体存储器的速度提高得要慢一些。而就速度而言相对提高最慢的是磁盘存储设备，它的访问时间仍在毫秒级。当然，这些设备在存储容量上有很大的提高。

高性能设备更贵一些，有时使用低成本设备进行并行操作也可以用合理的成本获得很高的性能。在第12章将看到如何使用多个普通处理器来实现高性能的多处理器计算机系统。多个磁盘驱动器也可以用来提供高性能的存储部件。

在1988年，在加州大学伯克利分校进行的研究提出了一个基于多个驱动器的存储系统，他们称之为RAID，表示廉价磁盘冗余阵列 (Redundant Array of Inexpensive Disk)。使用多个磁盘还可以提高系统的总体可靠性。提出了六种不同的配置结构，它们被称为RAID的层次，虽然这些并不涉及层次结构。

RAID 0是基本的结构，用来增强性能。一个单独的大文件被分成一些小块，把这些小块存到不同的磁盘上，从而实现把这个文件存储到不同的磁盘上，这称为数据条带化。当读取这个文件的时候，所有的磁盘可以并行传递它们的数据。文件的总传输时间等于一个单独磁盘的系统所需要的时间除以阵列中使用的磁盘个数，但是单个磁盘的访问时间，也就是用于定位数据起点的寻道和旋转延迟并没有减少。实际上，由于一个磁盘的操作独立于其他磁盘，所以访问时间各不一样，需要对访问的数据块进行缓冲，以便能把文件完整地组装起来，并把它作为一个单一的实体发送给请求的处理器。这是最简单的磁盘阵列操作，它只是提高了数据流的时间性能。

RAID 1用来提供更高的可靠性，它把数据的相同备份放到两个磁盘上，而不是一个。这两个磁盘相互称为彼此的镜像。那么如果一个磁盘出错了，所有的读写操作都指向它的镜像。这是用高昂的代价提高可靠性的方法，因为所有的磁盘都是冗余的。

RAID 2、RAID 3和RAID 4层次通过不同的奇偶校验方案来获得更高的可靠性，它们不需要整个磁盘的副本，所有的奇偶信息存储在一个磁盘上。

RAID 5也使用基于奇偶校验的错误恢复方案，但奇偶信息分布在各个磁盘中，而不是只在一个磁盘上。

后来一些混合方案被开发出来。例如，RAID 10是结合了RAID 0和RAID 1特征的阵列。对于RAID方案的更详细分析可以在参考文献[6~10]中找到。

RAID概念已经被商家接受，例如Dell公司提供基于RAID 0、RAID 1和RAID 5的产品。最后，我们应该注意到，在过去的几年里随着磁盘驱动器价格的大幅下降，在RAID中再说“廉价”磁盘可能不合适了。实际上，RAID已经被工业界重新定义为“独立”磁盘的代表。

### 商用磁盘考虑

大多数磁盘部件都设计成连到标准的总线上。一个磁盘部件的性能依赖于它的内部结构和

用于连接系统其他部分的接口。成本主要依赖于存储容量，但它也很大程度上受特定产品销量的影响。

**ATA/EIDE 磁盘** 使用最广泛的计算机是由IBM在1980年推出的个人计算机(PC)，就是大家熟知的IBM PC。用于连接到IBM PC总线的磁盘接口已经被开发出来，它的当前(增强)版本成为了一个标准，称为EIDE (Enhanced Integrated Drive Electronics, 增强型集成驱动器电路)或ATA (Advanced Technology Attachment, 高级技术配件)。很多磁盘制造商都生产有EIDE/ATA接口的磁盘，这种磁盘可以直接连到PCI总线(在4.7.1节讨论)，PCI总线用在很多PC上。实际上，Intel的Pentium芯片系列包含一个控制器，允许EIDE/ATA驱动器直接连到主板上。EIDE/ATA驱动器的一个重要优势是它们的低价格，这是因为它们是用于PC市场的。其主要缺点是如果同时使用两个驱动器来提高性能，每个驱动器都需要一个单独的控制器。

**SCSI磁盘** 就像我们在前面例子中解释的那样，很多磁盘都有一个用来连接标准SCSI总线的接口。这些磁盘更贵一些，但是它们表现出更好的性能，这是由SCSI总线相对于PCI总线的优势造成的。可以对多个磁盘驱动器同时访问，因为只有当驱动器准备好进行数据传输时，它的接口才真正连接到SCSI总线上。对于那些有很多小文件的访问请求应用来说，这一点特别有用，这种情况通常发生在用于文件服务器的机器上。

**RAID磁盘** RAID磁盘具有优良的性能，并能提供大容量高可靠性的存储。它们用在高性能计算机中，或用于那些需要更高可靠性级别的系统中。随着其价格降到更低的水平，它们在中等规模的计算机系统中也变得很有吸引力。

## 5.9.2 光盘

大容量存储设备也可以使用光学方法实现，人们熟知的用于音频系统的光盘(Compact Disk, CD)是这项技术的第一个实际应用。后来光学技术很快就用于计算机环境，以提供高容量的只读存储，被称为只读光盘(CD-ROM)。

[352]

第一代CD在20世纪80年代中期由索尼和菲利普公司开发出来，同时还发布了关于这些设备的完全规范。这个技术利用了可以使用数字来表示模拟声音信号的可能性。为了提供高质量的声音记录和再现，人们使用了16位的模拟声音信号采样，频率为每秒采样44 100次，这个采样频率是原来声音最高频率的两倍，从而可以精确地重建声音。CD可以保存最少一个小时的音乐，它的第一个版本被设计成可以保存75分钟的音乐，这需要大约 $3 \times 10^9$  (3G) 的存储量。从那以后，更高容量的设备被开发出来，一张视频CD可以存储一整部电影，这需要的存储容量比音频CD高将近一个数量级。多媒体CD同样适用于存储大量的计算机数据。

### CD技术

用于CD系统的光学技术以激光源为基础。一个激光束直射到旋转的盘面上，物理凹凸沿着光盘轨道排布，它们把射来的光束反射到一个光电探测器上，用它来探测所存的二进制模式。

激光器发出一束相关光，它准确地聚焦到盘面上。相关光由同步的波组成，它们有相同的波长。一束相关光与另一束相同类别的相关光混合，如果这两束光同相位，那么会产生一束更亮的光。但是，如果这两束光相位差180度，那么它们会互相抵消。因此，如果使用光电探测器来探测光束，在第一种情况下它会探测到一个亮点，在第二种情况下它会探测到一个暗点。

图5-32a显示了CD一小部分的横切面。最底层是聚碳酸酯塑料，起到透明盘基的作用。塑料表面可以通过刻出凹痕来编程存储数据，未刻的部分称为凸痕。然后在已编程的盘上覆上一个

很薄的铝反射层，再在铝反射层外包上一个丙烯酸保护层。最后，最外层被印上标签。光盘的总厚度是1.2毫米，这几乎都是聚碳酸脂塑料，其他层都非常薄。

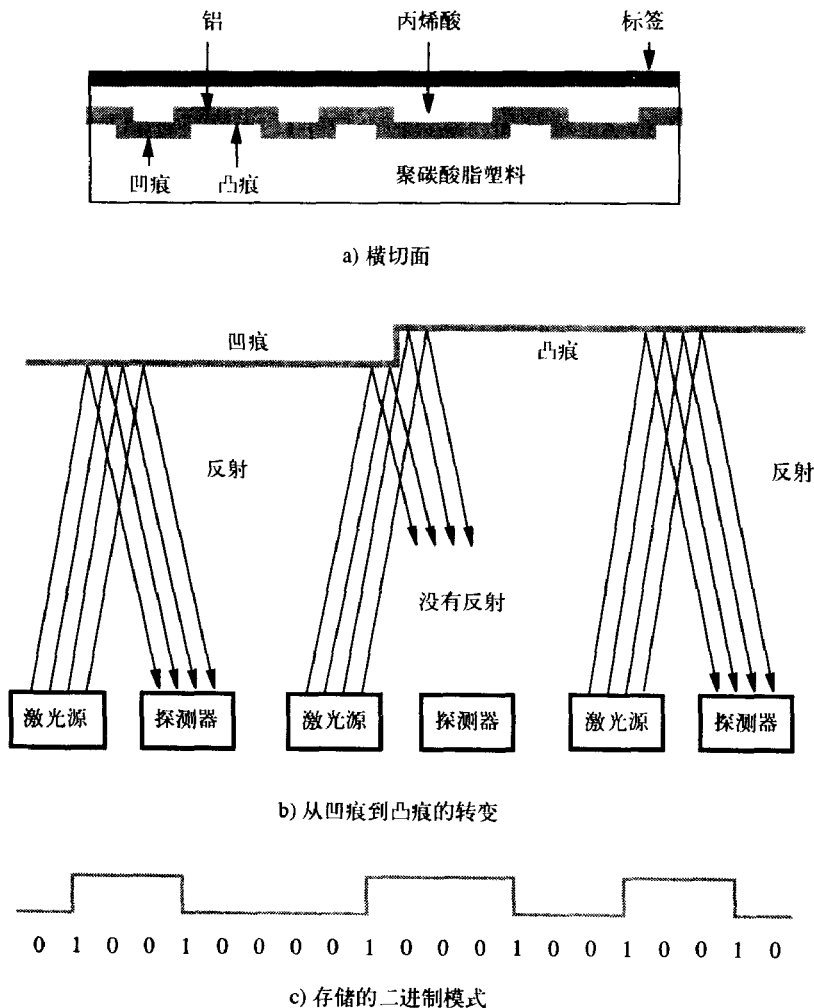


图5-32 光盘

激光源和光电探测器被放在聚碳酸脂的下方，发出的光线穿过塑料，在铝反射层上被反射回来，然后向回传播到光电探测器。注意从激光这边来看，凹痕相对凸痕来说实际上是一个凸起。

图5-32b显示了在激光束沿着盘面扫描并遇到一个从凹痕到凸痕的转变时发生的情况，图中显示了激光源和探测器的三个不同位置，它们随着光盘的转动而出现。当光只是在凹痕或凸痕上反射时，探测器将会看到反射的光，它会看到一个亮点。但是当光线移过从凹痕变成凸痕的边，或从凸痕到凹痕的边时，就会发生不同的情况。凹痕凹进光的波长的四分之一，因此从凹痕反射的波与从凸痕反射的波相位差180度，它们互相抵消。于是在凹痕-凸痕和凸痕-凹痕转换处探测器看不到反射的光波，它将探测到一个暗点。

图5-32c描述了在凸痕和凹痕之间的一些转换。如果被探测为暗点的凹凸转换表示二进制值1，平的部分表示0，那么探测到的二进制模式就如图所示。这个模式不直接表示存储的数据，CD使用一个复杂的编码方案来表示数据。每字节数据表示成14位编码，它提供了相当强的错误

检测能力。我们不再详细研究这个编码。

凹痕沿着盘面上的轨道排布。实际上只有一条物理轨道，它从盘的中间螺旋伸展到外边缘。但是，习惯上我们把每个跨越360度的圆形路径看作一个单独的轨道，这与磁盘使用的技术类似。CD的直径是120毫米，在中心有一个15毫米的孔。存储数据的轨道位于从半径25毫米到58毫米的区域中。轨道间的间隔是1.6微米，凹痕0.5微米宽，0.8到3微米长。一张盘上有超过15 000个的轨道。如果整个轨道螺旋全部展开，它的长度将超过5公里。

这些数字表明轨道密度约为6000轨道/厘米，它比磁盘的密度要高得多。硬盘轨道密度的范围是每厘米800到2000个轨道，软盘每厘米轨道数少于40个。

### CD-ROM

因为信息以二进制形式存储在CD上，所以它们也适合用作计算机系统的存储介质。最大的问题是如何保证数据的完整性。因为凹痕很小，所以很难完全正确地实现每一个凹痕。在音频和视频应用中，一些错误可以被容忍，因为它们对再现的声音和图像产生的影响不会被察觉到。然而，这样的错误在计算机应用中是不可接受的。由于物理上的缺陷是不可避免的，所以需要使用附加的位来提供错误检测和纠正能力。用在计算机应用中的CD有这种能力。它们被称为只读光盘（CD-ROM），因为在制造后它们的内容就只能被读取，像半导体只读存储器芯片一样。

存储的数据在CD-ROM的轨道上按块的形式组织，这些块称为扇区。扇区有几种不同的格式。一种格式称为模式1，它使用2 352字节的块。每个块有一个16字节的头，包含用于检测扇区起点的同步字段和用于标志扇区的地址信息。后面跟着的是2 048字节的存储数据。在扇区结尾有288个字节，用于实现错误纠正方案。每个轨道的扇区数各不相同，较长的外侧轨道有更多的扇区。

错误检测和纠正在多个层次上进行。就像在介绍CD时提到的，每个字节的存储信息使用14位的代码来编码，它有一定的纠错能力，能纠正单个位的错误。在短期脉冲中产生的错误影响多个位，这可以使用扇区结尾处的错误检查位来检测和纠正。

CD-ROM驱动器以很多种不同的旋转速度工作。基本速度称为1X，是每秒75个扇区，这提供了153 600字节/秒（150K字节/秒）的数据传输率。以这个速度和格式，使用存储75分钟音乐的标准光盘的CD-ROM有650M字节的存储容量。注意，驱动器速度只影响数据传输率，对光盘的存储容量没有影响。更高速度的CD-ROM驱动器相对基本速度标注。因此，一个40X的CD-ROM的数据传输率比1X的CD-ROM高40倍。注意这个传输速率（<6M字节/秒）要比硬盘的传输速率低得多，硬盘的传输速率在每秒几十兆的范围内。另一个大的性能区别是寻道时间，在CD-ROM中可能是几百毫秒。所以，就性能而言，CD-ROM显然要劣于磁盘，它们的吸引力在于物理尺寸小，成本低，还有容易作为可拆装和可移动的海量存储介质。

355

CD-ROM对计算机系统的重要性是因为与其他廉价便携介质（例如软盘和磁带）相比，它们有很大的存储容量和很快的访问速度，因此广泛应用于软件发布、数据库、大文本（书）、应用程序和视频游戏中。

### 可刻录CD

前面描述的CD是只读的设备，信息使用特殊的过程存储。首先，使用高功率激光在需要凹痕的位置烧出坑，以此来生产一个母盘。然后根据母盘生产一个模具，它在有坑的地方会有一个凸起。接着，把熔化的聚碳酸酯塑料注入模具，就可以生产出与母盘凹痕模式相同的CD了。这个过程显然只适合CD的批量生产。

20世纪90年代末年有一种新类型的CD被开发出来,计算机用户可以很容易地把数据刻到它上面,它被称为可刻录光盘(CD-Recordable, CD-R)。在生产过程中,在光盘上做出了一条螺旋轨道。在CD-R驱动器中使用激光在轨道上的有机染料上烧出凹痕来,当被烧的点加热到一定的温度时,它就变得不透明了。在以后读的时候,这个点反射较少的光。写入的数据被永久存储,未使用的部分可以在以后用来存储其他数据。

### 可擦写CD

最灵活的CD是那些可以由用户多次写入的CD,它们被称为可擦写光盘(CD-ReWritable, CD-RW)。

CD-RW的基本结构与CD-R类似。但是在可刻录层没有使用有机染料,而是使用银、铟、锡和碲的合金。这种合金在被加热和冷却时发生的反应非常有趣,也非常有用。如果它被加热到超过熔点(500摄氏度)然后被冷却,就变成非晶体状态,此时它吸收光线。但是如果它只被加热到200摄氏度,然后保持一段时间,就会发生称为退火的过程,导致合金变成晶体状态,此时它允许光线穿过。如果用晶体状态来表示凸痕区域,那么可以通过把选定点加热到超过熔点来创建凹痕。存储的数据可以使用退火过程来擦除,这使合金回到统一的晶体状态。反射材料放在刻录层的上面,在盘被读取的时候用来反光。

[356]

CD-RW驱动器使用三种不同功率的激光。最大功率用来刻录凹痕;中等功率用来把合金变成晶体状态,它被称为擦除功率;最低功率用来读取存储的数据。CD-RW光盘能被擦写的次数有一个限制,现在,这个次数可以达1000次。

CD-RW驱动器通常也能处理其他光盘介质,它能读取CD-ROM,能读写CD-R。它按标准互连接口的要求设计,例如EIDE、SCSI和USB。

CD-RW提供了低成本存储介质,它适合于信息档案存储,适用的范围可以从数据库到图形图像。也可以用于信息的小批量发布,与CD-R一样。现在CD-RW驱动器的速度已经足够快,可以用于每天的硬盘备份。CD-RW技术已经使得CD-R不那么重要了,因为它以稍高一点的成本提供了非常好的性能。

### DVD技术

CD技术的成功和对更大存储容量的不断寻求导致了DVD(Digital Versatile Disk, 数字多功能光盘)技术的发展。第一个DVD标准由一个公司联盟在1996年制定,目标是能在DVD盘的一面上存储一整部电影。

DVD光盘的物理尺寸与CD相同,盘体1.2毫米厚,直径120毫米。通过改变下面几项设计使得它的存储容量比CD高得多:

- 使用波长为635nm的红色激光器代替CD中使用的波长为780nm的红外激光器。波长的缩短可以把光线聚焦到一个更小的点上。
- 凹痕更小,最小长度0.4微米。
- 轨道更接近,轨道间距离是0.74微米。

使用了这些改进使得DVD的容量有4.7G字节。

使用两层或两面的盘可以进一步增加容量。单层单面的盘是在DVD-5标准中定义的,它的结构与图5-32a中的CD几乎一样。双层盘使用两个层,每层都刻有轨道。第一层是一个透明的盘基,与CD盘一样。但是它没有用铝来反射,而是在这层的凹痕和凸痕上覆盖一层半透明材料,

把它作为一个半反射体。然后在这层材料的表面上也刻上凹痕来编程存储数据。在第二层上的凹痕和凸痕上再放上一层放射材料。这种盘通过把激光束聚焦到需要的层上来实现读取。当激光聚焦到第一个层上，半透明材料反射回足够的光来检测存储的二进制模式；当激光聚焦到第二个层上，反射材料返回的光显示了这一层存储的信息。在两种情况下，没有被激光聚焦的层反射回少量的光，它被探测器电路当作噪声排除了。两个层的总存储容量是8.5G字节，在标准里这种盘称为DVD-9。

357

两个单面盘可以放在一起形成一个类似三明治的结构，其中上面的那张盘要翻个面。这可以使用单层盘来实现，像在DVD-10中规定的，组合成的盘的容量是9.4G字节。它也可以使用双层盘来实现，像在DVD-18中规定的，产生17G的存储容量。

DVD驱动器的访问时间跟CD驱动器相似，但是，由于DVD盘以相同的速度旋转，而它的凹痕密度更高，所以它的数据传输率要高得多。

### DVD-RAM

人们还开发出了一个DVD设备的可擦写版本，称为DVD-RAM，它提供了很大的存储容量。仅有的缺点是高价格和相对来说较慢的写速度。为了保证信息正确地记录在光盘上，人们使用了一个称为写验证的过程。这个过程是由DVD-RAM驱动器完成的，它读取存储的内容，然后把它与原始数据进行校对。

## 5.9.3 磁带系统

磁带适合于大量数据的离线存储，它们通常用于硬盘备份和归档存储。磁带记录时使用了与磁盘相同的原理，主要的区别是磁带的磁性薄膜镀在一条很细的宽度为0.5到0.25英寸的塑料带上。在磁带的宽度方向上平行记录了7到9位（对应一个字符），它与磁带的运动方向垂直。磁带每个位的位置都有一个单独的读写磁头，所以一个字符的所有位能并行读或写。字符的一个位用作奇偶位。

磁带上的数据按记录的形式组织，记录由间隙分开，如图5-33所示。磁带的运动只有在记录间隙位于读写磁头下方时才会停止。记录的间隙足够长，允许磁带在到达下一个记录的起点之前能达到正常速度。如果使用类似图5-29c中的编码方案在磁带上记录数据，那么记录间隙可以表示为没有磁性变化的区域。这允许记录间隙的检测独立于存储的数据。为了帮助用户组织大量数据，把一组相关的记录称为文件。文件的起点通过文件标记来识别，如图5-33所示。文件标记是一个特殊的单字符或多字符记录，它前面通常有一个比记录间间隙长一些的间隙。文件标记后的第一个记录可以用作这个文件的文件头或文件标志符，这允许用户在包含大量文件的磁带中搜索特定的文件。

358

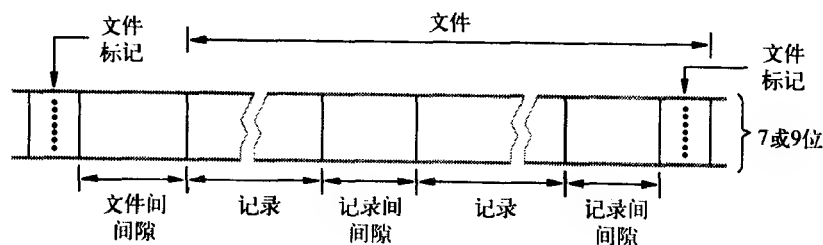


图5-33 磁带上的数据组织结构

磁带驱动器的控制器除了能执行读写命令，还能执行很多其他的控制命令。控制命令包括以下的操作：

- 回卷磁带
- 回卷并卸载磁带
- 擦除磁带
- 写磁带标记
- 前进一个记录
- 后退一个记录
- 前进一个文件
- 后退一个文件

在“写磁带标记”操作中的磁带标记与文件标记类似，只是它是用来标记磁带起点的。磁带结尾有时候使用EOT字符来标记（见附录E）。

有两种格式化磁带和使用磁带的方法。在第一种方法中，记录的长度是变化的，这可以有效地使用磁带，但是它不能更新和覆盖指定位置的记录。第二种方法使用固定长度的记录，这样它就允许更新指定的记录。虽然这可能看起来非常好，但是实际上并没有那么重要。磁带最通常的用途是备份硬盘信息和对数据归档存储，在这些应用中，磁带都是从头写到尾，所以记录的大小无关紧要。

#### 盒式磁带系统

磁带系统已经被开发用来为在线磁盘存储做备份。其中有一种这样的系统使用8毫米视频格式磁带，磁带位于一个盒中。这种部件称为盒式磁带，它们的存储容量为2G到5G字节，能以每秒几百K的速度传输数据。通过一个螺旋扫描系统横跨磁带进行操作来实现读写，这与盒式录像带驱动器类似。位密度能达到每平方英寸几千万比特。现在已经有能自动装盒和卸盒的多盒系统，所以数十G的在线存储备份可以不需要人来干涉。

### 5.10 结束语

在任何一种计算机中存储器都是一个主要的部件，它的容量和速度在决定计算机性能时起重要作用。在这一章中，我们介绍了存储器设计中最重要技术和组织结构细节。

半导体技术的发展使得存储器芯片的速度和容量有了很大的提高，每位成本也大幅下降。但是处理器芯片的发展更显著，特别是处理器芯片处理速度的提高使它的速度已经超过了存储器的速度。为了充分利用现代处理器的能力，计算机必须拥有很大而且很快的存储器。由于成本同样也很重要，所以存储器不能只是简单地使用快速的静态随机存储器来设计。就像我们在这章中看到的那样，这个问题的解决方案是使用存储器层次结构。

今天，可以提供的大容量主存都是使用动态存储器芯片实现的，这样的存储器就时钟周期而言可能比快速的处理器慢一个数量级，这需要使用一个SRAM高速缓存来减小处理器感受到的有效存储器访问时间。存储器延迟是决定计算机性能的一个重要参数，大量的研究工作都集中到开发用于最小化存储器延迟的方案上。我们描述了写缓冲区和预取技术如何减少了延迟的影响，它们在不需要执行高优先级存储器访问（由读失效引起）时可以执行紧急特性低一些的访问。如果同时访问一块连续字节，那么存储器延迟的影响还能减小，新型的存储器芯片正是

利用了这个特点进行开发。

磁盘或光盘形式的辅助存储设备提供了层次结构中最大的存储容量。虚拟存储器机制在磁盘和主存之间建立了对用户透明的交互。对虚拟存储器的硬件支持已经成为现代处理器的标准特征。

磁盘是计算机技术进步的一个很有意义的例子。它们通常是存储器层次结构中的低速部分。在不同时期,当新技术出现并有很好的发展前景时,磁盘继续存在的价值就会受到置疑。在20世纪80年代早期,“磁泡”存储器似乎会造成很大威胁,而近年来的竞争者是闪存驱动器和光盘。但是,磁盘技术不但没有被遗弃,反而不断发展。磁盘驱动器的容量不断变大,物理尺寸不断变小,而且每个存储位的成本也不断降低。

## 习题

- 5.1 对于一个使用 $512\text{K} \times 8$ 存储器芯片的 $8\text{M} \times 32$ 存储器,给出它的类似图5-10的框图。
- 5.2 考虑图5-6中的动态存储器单元。假设 $C = 50$ 飞法( $10^{-15}$ 法),通过晶体管的泄漏电流大约是9皮安( $10^{-12}$ 安),电容器充满电荷的时候电压为4.5伏。这个单元必须在电压降到3伏之前进行刷新。估算最小刷新率。
- 5.3 在图5-8的右下角有数据输入和数据输出寄存器。画出能实现每个寄存器中一位的电路,并显示出它到一侧的“读/写电路和锁存器”块和到另一侧的数据总线所需要的连接。
- 5.4 考虑由SDRAM芯片构成的主存,对芯片的时序要求与图5-9所示的基本一致,只是脉冲长度为8。假设并行传输32位数据,如果使用133MHz的时钟,传输下面数据需要多少时间:
- (a) 32字节数据
- (b) 64字节数据
- 每个高速缓存的延迟是多少?
- 5.5 分析下面这句话为什么不对:“使用更快的处理器芯片可以相应地提高计算机的性能,尽管主存的速度保持不变。”
- 5.6 一个程序包含两个嵌套的循环——较小的内部循环和较大的外部循环。程序的一般结构在图P5-1中给出,图中十进制的地址显示了两个循环的位置和整个程序的起点和终点,在存储器不同部分(17~22、23~164、165~239等)的所有位置都包含顺序执行的指令序列。这个程序运行在按直接映射方法组织指令高速缓存的计算机上,并且各个参数如下:

主存大小	64K字
高速缓存大小	1K字
块大小	128个字

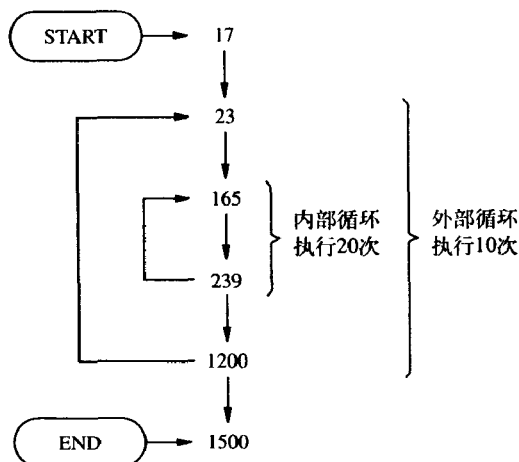
主存的周期时间是 $10\tau_s$ ,高速缓存的周期时间是 $1\tau_s$ 。

- (a) 在主存地址中指定TAG、BLOCK和WORD字段的位数。
- (b) 计算在图P5-1中的程序的执行期间取指令需要的总时间。
- 5.7 一个计算机在主存和处理器之间使用一个小的直接映射高速缓存。高速缓存有4个16位字,每个字有一个相联的13位的标志,如图P5-2a所示。当读操作期间发生一次失效时,把请求的字从主存读出,然后送到处理器。同时它被拷贝到高速缓存中,并且其块号被存到相联的标志中。考虑下面程序中的循环,其中所有的指令和操作数都是16位长。

360

361





图P5-1 习题5.6的程序结构

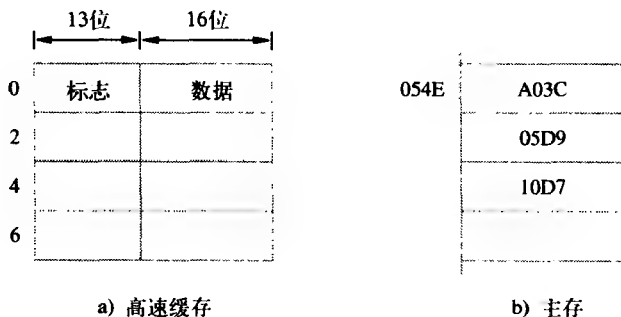
```

LOOP   Add      (R1) +, R0
        Decrement R2
        BNE     LOOP
  
```

假设在进入循环前，寄存器R0、R1和R2包含的内容分别为0、054E和3。再假设主存中包含图P5-2b所示的数据，其中所有的入口都以16进制给出。循环在LOOP=02EC处开始。

(a) 显示每执行完一次循环后高速缓存中的内容。

(b) 假设主存访问时间是 $10\tau$ ，高速缓存是 $1\tau$ ，计算每次循环的执行时间。不考虑在存储器周期期间处理器耗费的时间。



a) 高速缓存

b) 主存

图P5-2 习题5.7中高速缓存和主存中的内容

- 362** 5.8 重复题5.7，假设在高速缓存中只存一条指令，数据操作数直接从主存中获取，并且不拷贝到高速缓存中。为什么这种选择比把指令和数据都写到高速缓存中速度更快？
- 5.9 一个组相联高速缓存包含64个块，每组4块。主存有4096个块，每个块有128个字。
- (a) 主存地址有多少位？
- (b) 在每个TAG、SET和WORD字段中各有多少位？
- 5.10 一个计算机系统的主存由1M16位字组成，它还有4K字的高速缓存，按组相联组织，每组4个块，每块64个字。
- (a) 计算每个主存地址的TAG、SET和WORD字段的位数。

(b) 假设高速缓存初始为空, 处理器按顺序从位置0、1、2、...、4351取4352个字, 然后它再重复这个取操作序列9次。如果高速缓存比主存快十倍, 估算使用高速缓存后性能提高的倍数。假设在块替换时使用LRU算法。

5.11 假设只要从主存中读出一个新块且它在高速缓存中对应的组已经满了, 新块就替换这个组中最近使用的块。现在重复题5.10。

5.12 5.5.3节用图5-19中的程序描述了不同高速缓存映射技术的效果。假设这个程序的第二个循环处理元素的顺序改成与第一个循环相同, 也就是使用如下语句控制第二个循环

for  $i := 0$  to 9 do

推算出图5-20到图5-22针对这个程序的等式。从此可以得出什么结论?

5.13 一个字节可编址的计算机有一个能容纳8个32位字的小数据高速缓存, 每个高速缓存块有1个字。当执行一个给定的程序的时候, 处理器按下列地址顺序读取数据:

200, 204, 208, 20C, 2F4, 2F0, 200, 204, 218, 21C, 24C, 2F4

这个模式重复四遍。

(a) 如果使用直接映射方式, 显示每次循环结束时高速缓存中的内容。计算这个例子的命中率。假设高速缓存初始为空。

(b) 如果换成采用LRU替换算法的相联映射高速缓存, 重复第一问。

(c) 如果是4路组相联高速缓存, 重复第一问。

363

5.14 假设每个高速缓存块包含两个32位字, 重复题5.13。在第三问中, 使用2路组相联高速缓存。

5.15 在图5-25b中的交叉存储器系统中, 在设计这个系统的高速缓存时,  $k$ 值是如何影响块大小的?

5.16 在很多计算机中, 高速缓存块的大小在32字节到128字节之间的范围内。使用更大或更小的高速缓存块的主要优点和缺点各是什么?

5.17 就高速缓存块的大小考虑交叉的效果。使用类似于5.6.2节中的计算来估计块大小为16个字、8个字和4个字时性能的提高。假设所有被装入的字至少被处理器访问一次。

5.18 假设计算机有L1和L2高速缓存, 就像5.6.3节中讨论的一样, 高速缓存块由8个字组成。假设两个高速缓存的命中率一样, 都是指令为0.95, 数据为0.90, 再假设在这两个高速缓存中访问一个8个字的块需要的时间分别为 $C_1 = 1$ 周期和 $C_2 = 10$ 周期。

(a) 如果主存使用交叉, 那么处理器感受到的平均访问时间是多少? 假设存储器访问使用与5.6.1节中一样的参数。

(b) 如果主存不交叉, 平均访问时间是多少?

(c) 交叉获得的性能提高为多少?

5.19 假设每个高速缓存块有4个字, 重复题5.18。假设L2高速缓存使用SRAM芯片实现, 为 $C_2$ 估计一个合理的值。

5.20 考虑下面对高速缓存概念的类比。一个修理员到一个住宅修理加热系统。他带了一个工具箱, 里面有他最近在类似工作中使用的工具。他反复使用这些工具, 直到需要另一个工具。可能在屋外的卡车上他有他需要的工具, 如果卡车上没有, 他就必须回店铺中去拿。假设我们把工具箱、卡车和店铺对应成L1高速缓存、L2高速缓存和计算机主存, 那么这个类比是否恰当? 讨论它正确和不正确的地方。

5.21 一个由32位数字组成的 $1024 \times 1024$ 数组按以下方法标准化。对每一列, 找到最大的元素, 这个列中的所有元素除以这个最大元素。假设虚拟存储器中每页包含4K字节, 并且在这个

计算中主存的1M字节被分配用来存储数据。假定当发生页故障时从磁盘装载一个页到主存需要40毫秒。

364

(a) 如果数组元素按列顺序存储在虚拟存储器中, 会发生多少次页故障?

(b) 如果元素按行顺序存储会发生多少次页故障?

(c) 对于(a)和(b)中的布局估算标准化过程总共需要多少时间。

5.22 考虑一个计算机系统, 它物理内存的可用页在几个应用程序中分配。当分配给一个程序的所有页都满了, 而且需要一个新页时, 新页必须替换出一个已经存在的页。操作系统监视页传送活动并动态调整对不同程序的页分配。给出一个能被操作系统用来最小化的总体页传送率的合适策略。

5.23 在一个有虚拟存储器的计算机中, 一条指令的执行可能被页故障中断。为了能使指令在以后继续执行, 需要保存哪些状态信息? 注意把一个新页装入主存涉及DMA传输, 它需要执行其他指令。放弃被中断的指令, 然后把它完全重新执行一遍是否更简单? 重新执行一遍能实现吗?

5.24 当一个程序产生一个对不在物理主存中的页引用时, 这个程序的执行被挂起, 直到请求的页被装入主存。当一个页中指令的操作数在另一个页中时, 会有什么困难发生? 为了处理这种情况, 处理器必须拥有何种能力?

5.25 一个磁盘部件有24个记录面, 它共有14 000个柱, 平均每个磁带有400个扇区, 每个扇区包含512字节的数据。

(a) 这个磁盘最多能存储多少字节?

(b) 以7200rpm的速度旋转时, 数据传输率是每秒多少字节?

(c) 假设每个扇区有512字节, 如果使用32位的字, 给出一个指定磁盘地址的合理方案。

5.26 对多数磁盘传输来说, 在磁盘上访问特定的数据块时, 寻道时间加上旋转延迟通常比数据传送周期长得多。考虑对在5.9.1节中作为例子给出的3.5英寸磁盘的连续多次访问, 不论是读操作还是写操作, 访问的块的平均长度都是8K字节。

(a) 假设这些块随机分布在磁盘上, 估算寻道时间和旋转延迟占总时间的平均百分比。

(b) 重新排列这些磁盘访问, 使得在90%的情况中下一次访问的数据块与这次访问的数据块在同一个柱上。在这种情况下重复第一问。

5.27 磁盘系统的平均寻道时间和旋转延迟分别是6ms和3ms, 向或从磁盘传输数据的速率是每秒30M字节, 并且所有的磁盘访问都是对8K字节数据的访问。磁盘DMA控制器、处理器和主存都连到一个单一的总线上, 总线数据宽度为32位, 向或从主存的一次总线传输花费10纳秒。

365

(a) 能同时向或从主存传输数据的磁盘部件的最大数目是多少?

(b) 在一个长时间段内有一系列独立的8K字节传输发生, 在这段时间内平均有百分之几的主存周期被磁盘部件窃取?

5.28 假定在虚拟存储器系统中用磁盘作为辅助存储设备, 用来存储程序和数据文件, 哪一种磁盘参数将对页大小的选择产生影响?

5.29 一个磁带驱动器有以下参数:

位密度	2000bits/cm
磁带速度	800cm/s
翻转运动方向的时间	225ms

在记录间间隙上花费的最短时间	3ms
平均记录长度	4000个字符

如果它具有在前进和后退两个方向上读取记录的能力, 估算这个能力使得时间缩短的百分比是多少。假设记录被随机访问, 连续两次访问的两条记录间的平均距离是4条记录。

## 参考文献

1. T.C. Mowry, "Tolerating Latency through Software-Controlled Data Prefetching," *Tech. Report CSL-TR-94-628*, Stanford University, Calif., 1994.
2. J.L. Baer and T.F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proceedings of Supercomputing '91*, 1991, pp. 176-186.
3. J.W.C. Fu and J.H. Patel, "Stride Directed Prefetching in Scalar Processors," *Proceedings of the 24th International Symposium on Microarchitecture*, 1992, pp. 102-110.
4. D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981, pp. 81-85.
5. D.A. Patterson, G.A. Gibson, and R.H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988, pp. 109-166.
6. P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, and D.A. Patterson, "RAID: High Performance, Reliable Secondary Storage," *ACM Computing Surveys*, vol. 26, no. 2, June 1994, pp. 145-185.
7. D.A. Patterson and J.L. Hennessy, *Computer Architecture — A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, CA, 1996.
8. A.S. Tannenbaum, *Structured Computer Organization*, 4th ed. Prentice Hall, Upper Saddle River, NJ, 1999.
9. "RAID Technology White Paper," Dell Computer Corporation, 1999.
10. A. Clements, *The Principles of Computer Hardware*, 3rd ed., Oxford University Press, 2000.



## 本章目标

在本章中你将学习以下内容:

- 使用超前进位逻辑同步产生进位信号且以层次结构实现的高速加法器
- Booth算法, 用来在有符号数乘法中根据乘数的位模式选择被乘数求和项
- 高速乘法器, 使用进位保留加法并行加入求和项
- 除法运算电路
- 浮点数的IEEE标准表示形式以及浮点数的基本算术运算

367

所有数字计算机的基本运算操作都是两个数的加和减。算术运算属于机器指令层次。与第1章讲述的基本逻辑功能, 比如AND、OR、NOT及XOR(异或)相同, 算术运算也在处理器的算术逻辑部件(ALU)子系统中实现。本章将介绍实现算术运算的逻辑电路。执行加法操作所需要的时间会影响处理器的性能。与加减操作相比, 乘除操作需要更复杂的电路, 并且也会影响处理器的性能。我们将介绍几种现代计算机使用的进行高速算术运算的技术。

与算术运算相比, 逻辑运算很容易使用组合电路实现。它们只需要在操作数的每一位上进行独立的布尔运算, 而算术运算则还需要进位/借位等辅助信号。

我们在2.1节中描述了有符号二进制数的表示, 还指出了对于执行加减运算来说, 补码是最好的表示方法。在图2-4的例子中我们看到, 两个 $n$ 位有符号数可以使用 $n$ 位二进制加法相加, 将符号位用与其他位一样的方法进行处理。换句话说, 计算无符号二进制数加法的逻辑电路也可以用来计算补码表示的有符号数的加法。如果没有溢出, 则加法的和是正确的, 且可以忽略任何进位输出。本章的前两节将介绍加法与减法的逻辑电路网络。

## 6.1 有符号数加减法

图6-1显示了将两个数 $X$ 、 $Y$ 中同等权重的两位 $x_i$ 、 $y_i$ 相加时, 求和函数与进位输出函数的逻辑真值表。图中还列出了这两个函数的逻辑表达式, 以及两个4位无符号数7和6相加的例子。请注意加法过程中的每一步都必须包含一个进位输入位。我们用 $c_i$ 表示第 $i$ 步的进位输入, 同时它也是第 $(i-1)$ 步的进位输出。

图6-1中 $s_i$ 的逻辑表达式可以用三输入端异或门实现, 在图6-2a中它是单步二进制加法所需逻辑的一部分。进位输出函数 $c_{i+1}$ , 使用两级与-或逻辑电路实现。图中还使用了一个方便的符

号——全加器 (full adder, FA) 来表示单步加法所需的全部逻辑电路。

图6-2b显示了  $n$  个全加器部件的级联, 可以用来计算两个  $n$  位数的加法。由于进位要在级联中传递或波动, 这种结构被称为  $n$  位行波进位加法器。

368 最低有效位 (least-significant-bit, LSB) 位置上的进位输入  $c_0$  提供了为数字加1的简单方法。例如, 得到一个数的补码需要对该数的反码加1。也可以通过进位信号将  $k$  个加法器互连, 从而形成可以处理  $kn$  位输入数字的加法器, 如图6-2c所示。

$x_i$	$y_i$	进位输入 $c_i$	和 $s_i$	进位输出 $c_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \bar{x}_i \bar{y}_i c_i + \bar{x}_i y_i \bar{c}_i + x_i \bar{y}_i \bar{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

示例:

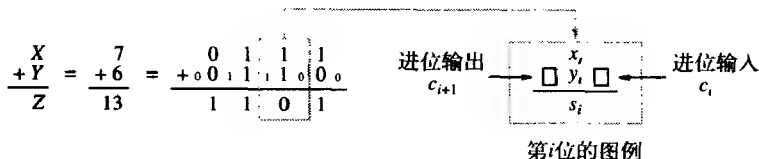


图6-1 单步二进制加法的逻辑说明

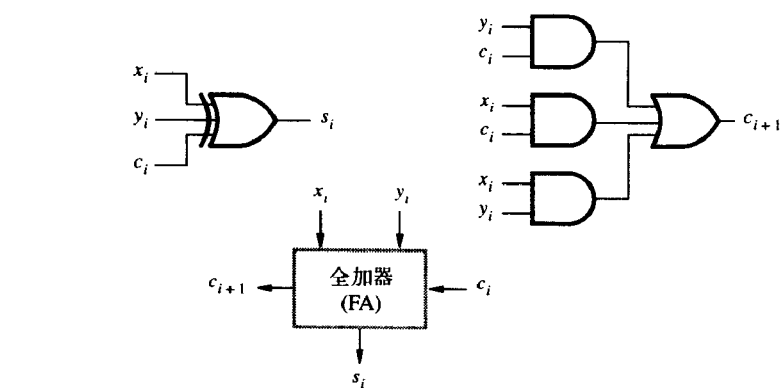
### 加法/减法逻辑单元

图6-2b中的  $n$  位加法器可以计算补码数  $X$ 、 $Y$  的加法, 其中  $x_{n-1}$  与  $y_{n-1}$  位为符号位。这时, 进位输出位  $c_n$  并不是加法结果的一部分。2.1.4节讨论了算术溢出。只有当两个操作数的符号相同时溢出才可能发生。这时如果计算结果的符号改变, 则表明是发生了溢出。因此, 可以为  $n$  位加法器添加执行以下逻辑表达式的溢出检测电路:

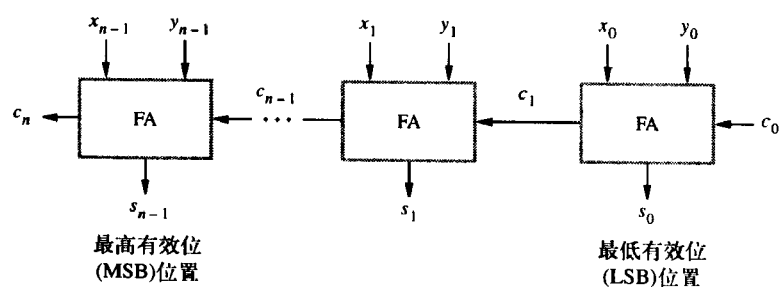
$$\text{溢出} = x_{n-1} y_{n-1} \bar{s}_{n-1} + \bar{x}_{n-1} \bar{y}_{n-1} s_{n-1}$$

还可以证明当进位位  $c_n$  与  $c_{n-1}$  不同时就发生了溢出 (参见习题6.9)。因此, 使用一个异或门执行表达式  $c_n \oplus c_{n-1}$  就可以得到更简单的溢出检测电路。

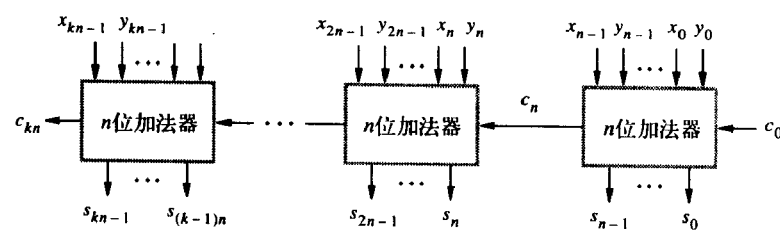
369 为了执行补码数  $X$  与  $Y$  的减法操作, 我们先得到  $Y$  的补码并将其与  $X$  相加。图6-3所示的逻辑电路网络可以根据加法/减法输入控制线的值执行加法或减法操作。执行加法时控制线置0, 将  $Y$  向量不加改变地送至加法器的一个输入端, 并使进位输入信号  $c_0$  为0。当加/减控制线置1时,  $Y$  向量通过异或门形成反码 (即按位取反), 并且  $c_0$  置1从而得到  $Y$  的补码。记住, 负数补码的计算方法与正数是完全相同的。可以在图6-3中添加一个异或门来检测溢出条件  $c_n \oplus c_{n-1}$ 。



a) 单步逻辑



b)  $n$ 位行波进位加法器



c)  $k$ 个 $n$ 位加法器的级联

图6-2 二进制向量的加法逻辑

370

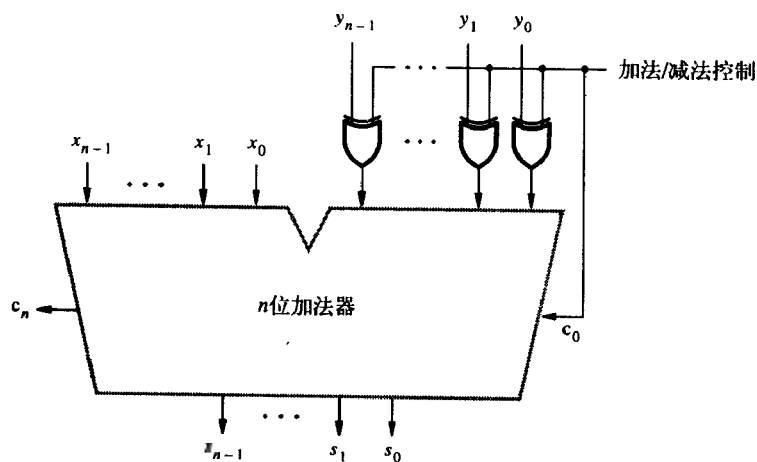


图6-3 二进制加法-减法逻辑网络



## 6.2 快速加法器设计

如果图6-3中的加法/减法部件使用 $n$ 位的行波进位加法器,可能需要经历很长的延迟才能得到输出 $s_0$ 至 $s_{n-1}$ 以及 $c_n$ 。这个延迟是否可以被接受只取决于其他处理器部件的速度、寄存器和高速缓存的数据传输时间的要求。一个逻辑门电路所需的延迟依赖于构造该电路的集成电路的电子工艺(见附录A),以及从输入至输出所经历的逻辑门的数目。通过任何一个以特定工艺制造的门构成的组合逻辑网络所需的延迟,是该网络中最长信号传输路径上所有逻辑门延迟的总和。

对于 $n$ 位行波进位加法器,最长路径是从最低有效位LSB位置上的输入 $x_0$ 、 $y_0$ 及 $c_0$ 到最高有效位(most-significant-bit, MSB)位置上的输出 $c_n$ 与 $s_{n-1}$ 。

371

使用图6-2a所示的逻辑实现, $c_{n-1}$ 在 $2(n-1)$ 个门延迟后得到, $s_{n-1}$ 需要再经过一个异或门延迟才正确。而最终的进位输出 $c_n$ 需要经历 $2n$ 个门延迟。因此,如果使用行波进位加法器实现图6-3中的加法/减法部件,可以在 $2n$ 个门延迟后得到所有的求和位,包括 $Y$ 输入端经历的异或门延迟。如果再使用 $c_n \oplus c_{n-1}$ 来检测溢出,则检测结果可以在 $2n+2$ 个门延迟后得到。

有两种方法可以减少加法器的延迟。第一种方法是使用最快的电子工艺实现行波进位及其变形的逻辑设计。第二种方法是使用比图6-2b更大的逻辑门网络结构。下一小节将讲述第二种方法中比较容易理解的一个版本。实际上有许多技术已经被用来实现高速加法器。它们包括快速传播进位信号的电子电路设计,以及下一小节中的基本网络结构的各种变形。

### 超前进位加法

快速加法器电路必须加速进位信号的产生。第 $i$ 步 $s_i$ (和)与 $c_{i+1}$ (进位输出)的逻辑表达式为(参见图6-1)

$$s_i = x_i \oplus y_i \oplus c_i$$

和

$$c_{i+1} = x_i y_i + x_i c_i + y_i c_i$$

将第二个等式因式分解为

$$c_{i+1} = x_i y_i + (x_i + y_i) c_i$$

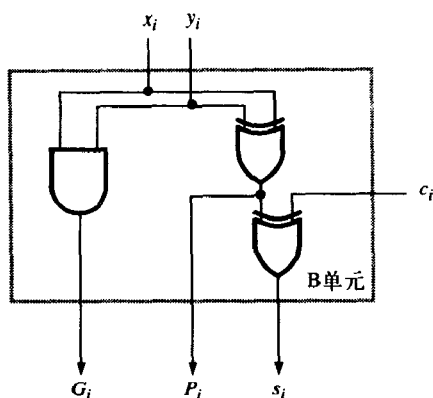
因此我们可以写出

$$c_{i+1} = G_i + P_i c_i$$

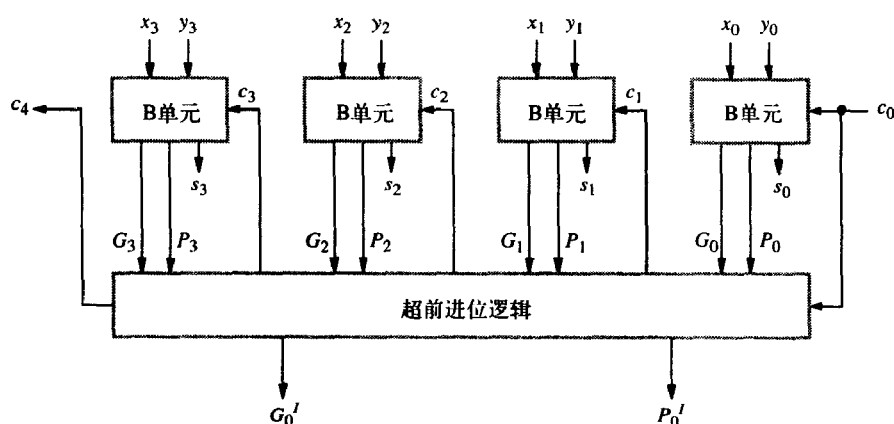
其中

$$G_i = x_i y_i \text{ 以及 } P_i = x_i + y_i$$

表达式 $G_i$ 和 $P_i$ 分别称为第 $i$ 步的生成函数与传播(propagate)函数。如果第 $i$ 步的生成函数等于1,则 $c_{i+1} = 1$ ,而与进位输入 $c_i$ 无关。这种情况发生在 $x_i$ 与 $y_i$ 都为1时。传播函数意味着当 $x_i$ 或者 $y_i$ 其中之一为1时,进位输入就会产生进位输出。所有的 $G_i$ 与 $P_i$ 都可以在 $X$ 与 $Y$ 向量加载到 $n$ 位加法器输入端一个逻辑门延迟后独立并行地生成。每一位包括一个与门以生成 $G_i$ ,一个或门以生成 $P_i$ ,以及一个三输入端异或门以生成 $s_i$ 。仔细观察会得出更简单的电路,我们发现使用 $P_i = x_i \oplus y_i$ 足可以实现传播函数。这个公式只在 $x_i = y_i = 1$ 时与 $P_i = x_i + y_i$ 不同。但这时 $G_i = 1$ ,所以 $P_i$ 是0是1无关紧要。这样,使用两个两输入端异或门的级联来实现三输入端异或功能,图6-4a中的基本单元B可以用于每一位段的计算。



a) 位段单元



b) 4位加法器

图6-4 4位超前进位加法器

使用以 $i-1$ 为下标的变量扩展 $c_i$ ，并将其代入 $c_{i+1}$ 表达式，我们得到

372

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

继续这种扩展，任一进位变量的最终表达式为

$$c_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i P_{i-1} \cdots P_1 G_0 + P_i P_{i-1} \cdots P_0 c_0 \quad (6-1)$$

这样，所有进位都可以在加载输入信号 $X$ 、 $Y$ 及 $c_0$ 三个门延迟后得到，因为生成所有 $P_i$ 与 $G_i$ 信号只需一个门延迟，再加上 $c_{i+1}$ 与-或电路中的两个门延迟。因此， $n$ 位加法过程只需四个门延迟，而与 $n$ 无关。

我们现在考虑一下4位加法器的设计。其进位可以实现为

$$c_1 = G_0 + P_0 c_0$$

$$c_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$c_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

图6-4b显示了完整的4位加法器。进位在标记为超前进位逻辑的部件中实现。以这种形式实现的加法器称为超前进位加法器。该加法器对所有的进位位有三个门延迟，对所有的求和位有四个门延迟。相比较而言，在4位行波进位加法器中， $s_3$ 需要七个门延迟，而 $c_4$ 需要八个门延迟。

373

如果想要扩展图6-4b的超前进位加法器，使它能够对更长的操作数进行加法运算，就会碰到门的扇入 (fan-in, 即输入端数目) 限制问题。由表达式6-1，我们看到最后的与门和或门生成  $c_{i+1}$  时需要的扇入为  $i+2$ 。在4位加法器中， $c_4$  需要的扇入为5。这已经接近实际门的极限了。所以图6-4b中的加法器设计不能直接对长操作数进行扩展。但是，如果我们将几个4位加法器级联起来，如图6-2c所示，就可以构造出较长的加法器了。

8个4位超前进位加法器可以如图6-2c那样连接起来构成一个32位加法器。级联中高端的4位加法器生成求和位  $s_{31}$ 、 $s_{30}$ 、 $s_{29}$ 、 $s_{28}$  以及  $c_{32}$  所需的延迟计算如下：低端加法器的进位输出  $c_4$  在输入操作数  $X$ 、 $Y$  及  $c_0$  加载到32位加法器三个门延迟后获得。之后，第二个加法器的输出  $c_8$  再经过两个门延迟获得， $c_{12}$  需要再经过两个门延迟，依次类推。最后，高端4位加法器的进位输入  $c_{28}$  可以在  $(6 \times 2) + 3 = 15$  个门延迟后获得。这样， $c_{32}$  和高端加法器内的所有进位还需要两个门延迟，而4个求和位需要再经历一个门延迟获得，一共是18个门延迟。作为对比，如果使用行波进位加法器， $s_{31}$  与  $c_{32}$  的总延迟数为63和64。

下一小节将对刚才讨论的级联结构加以改进，从而进一步减少延迟。关键的思想是并行地生成进位  $c_4$ 、 $c_8$  等，就像在4位超前进位加法器中并行地生成  $c_1$ 、 $c_2$ 、 $c_3$ 、 $c_4$  一样。

### 高层的生成与传播函数

在刚才讨论的32位加法器中，进位  $c_4$ 、 $c_8$ 、 $c_{12}$  等波动地通过各个4位加法器部件，每个部件两个门延迟，这与行波进位加法器中单个进位波动通过每个位段的方式非常相像。通过使用高层的生成与传播函数，就有可能在高层的超前进位电路中使用超前进位方法并行地生成进位  $c_4$ 、 $c_8$ 、 $c_{12}$  等。

374

图6-5显示了由四个4位加法器部件构成的16位加法器。这些部件提供了新的输出函数  $G'_k$  与  $P'_k$ ，其中  $k=0$  对应于第一个4位部件（如图6-4b）， $k=1$  对应于第二个4位部件，依次类推。在第一个部件中，

$$P'_0 = P_3 P_2 P_1 P_0$$

$$G'_0 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

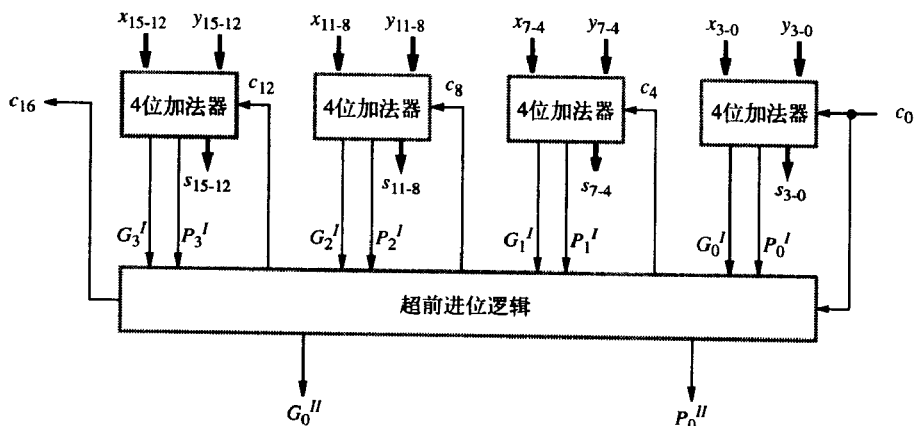


图6-5 由4位加法器构造的16位超前进位加法器（见图6-4b）

简单地说, 第一层 $G_i$ 与 $P_i$ 函数决定了位段 $i$ 是否生成或传播进位, 而第二层 $G'_i$ 与 $P'_i$ 函数决定了部件 $k$ 是否生成或传播进位。有了这些新函数, 就没有必要等待进位波动通过4位部件了。进位 $c_{16}$ 由图6-5所示的超前进位电路按以下公式生成

$$c_{16} = G'_3 + P'_3 G'_2 + P'_3 P'_2 G'_1 + P'_3 P'_2 P'_1 G'_0 + P'_3 P'_2 P'_1 P'_0 c_0$$

4位部件的进位输入也由相似的简单表达式并行地生成。 $c_{16}$ 、 $c_{12}$ 、 $c_8$ 及 $c_4$ 的表达式形式分别与6-4b超前进位电路所实现的 $c_4$ 、 $c_3$ 、 $c_2$ 及 $c_1$ 相同。只是变量名改变了。因此, 图6-5中超前进位电路的结构与图6-4b是完全相同的。但是我们应当注意, 由4位加法器部件内部产生的进位 $c_4$ 、 $c_8$ 、 $c_{12}$ 、 $c_{16}$ 在图6-5中并不需要, 因为在这里它们是由高层的超前进位电路生成的。

现在, 我们考虑16位超前进位加法器产生输出所需的延迟。超前进位电路产生进位所需的延迟比生成 $G'_i$ 和 $P'_i$ 多两个门延迟。而生成 $G'_i$ 和 $P'_i$ 在生成 $G_i$ 和 $P_i$ 之后分别需要两个和一个门延迟。因此, 超前进位电路在 $X$ 、 $Y$ 及 $c_0$ 加载到输入端五个门延迟后即可得到所有的进位。进位 $c_{15}$ 是在图6-5的高端4位部件内部产生的, 它比 $c_{12}$ 晚两个门延迟, 之后 $s_{15}$ 又晚了一个门延迟。因此,  $s_{15}$ 将在八个门延迟后得到。注意, 如果采用级联4位超前进位加法器的方法构造16位加法器, 生成 $c_{16}$ 和 $s_{15}$ 的延迟分别为九和十个门延迟, 而图6-5只需五和八个门延迟。

375

两个16位加法器部件可以级联构成一个32位加法器。这时, 低端部件的输出 $c_{16}$ 成为高端部件的进位输入。其延迟要比以前讨论过的32位加法器少得多, 那时我们通过级联八个4位加法器来构造32位加法器。在该加法器中,  $s_{31}$ 在18个门延迟后得到,  $c_{32}$ 在17个门延迟后得到。级联两个16位加法器的延迟分析如下: 我们刚刚讨论过, 低端部件的进位输出 $c_{16}$ 在五个门延迟后得到。高端部件的 $c_{28}$ 和 $c_{32}$ 在随后的两个门延迟得到,  $c_{31}$ 在 $c_{28}$ 之后的两个门延迟得到。因此 $c_{31}$ 在总共九个门延迟后得到, 而 $s_{31}$ 则需要十个门延迟。另外,  $s_{31}$ 和 $c_{32}$ 分别在十和七个门延迟后获得, 作为比较, 如果使用八个4位加法器级联构成的32位加法器, 获得相同的结果则分别需要18和17个门延迟。

从第一层 $G_i$ 和 $P_i$ 函数生成第二层 $G'_i$ 和 $P'_i$ 函数所用的推理可以用来从 $G'_i$ 和 $P'_i$ 函数生成第三层 $G''_i$ 和 $P''_i$ 函数。这两个第三层函数在图6-5中被表示为超前进位逻辑的输出。使用四个图6-5中的16位加法器, 再加上产生进位 $c_{16}$ 、 $c_{32}$ 、 $c_{48}$ 、 $c_{64}$ 的超前进位逻辑电路, 就可以构成一个64位加法器。使用上面对16位加法器推理的扩展, 可以证明这个加法器的延迟为 $s_{63}$ 需要12个门延迟,  $c_{64}$ 需要七个门延迟。(参见习题6.10。)

### 6.3 正数乘法

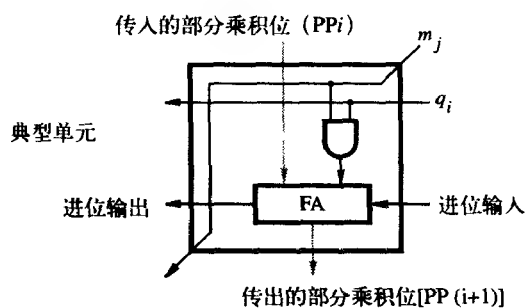
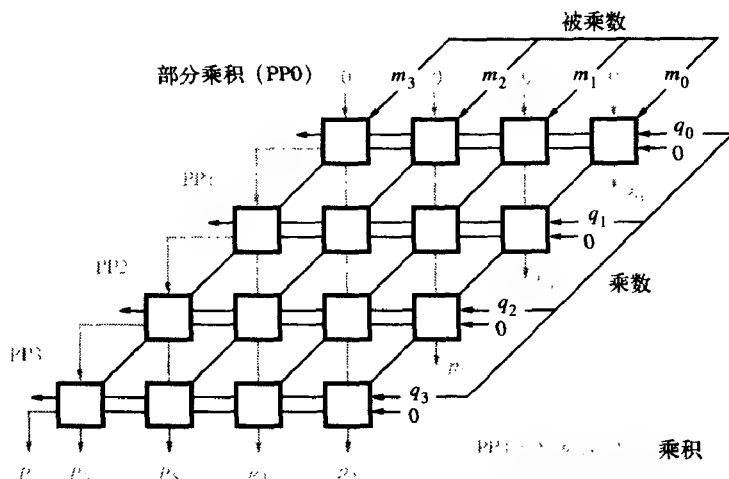
图6-6a显示了二进制系统中手工进行整数乘法的普通算法。该算法适用于无符号数和有符号正数。两个 $n$ 位数的积少于 $2n$ 位, 因此如图所示, 本例中两个4位数的积为8位。在二进制系统中, 被乘数与乘数的一位相乘是很容易的。如果乘数位为1, 则在适当的位置放入被乘数, 并添加到部分积。如果乘数位为0, 则放入0, 如示例的第三行。

376

正数的二进制乘法可以用二维组合逻辑阵列实现, 如图6-6b所示。每个单元的主要组成部分是一个全加器FA。每个单元的与门根据乘数位 $q_i$ 的值决定被乘数位 $m_j$ 是否要加到传入的部分乘积位。如果 $q_i = 1$ , 每一行 $i$  (其中 $0 < i < 3$ ) 将 (经过适当移位的) 被乘数加到传入的部分乘积 $PP_i$ 上, 以生成传出的部分乘积 $PP(i+1)$ 。如果 $q_i = 0$ , 则 $PP_i$ 就不经改变地向下传出。 $PP_0$ 为全0, 而 $PP_4$ 为乘积结果。被乘数沿着倾斜的信号传输路径每行左移一个位置。

$$\begin{array}{r}
 1101 \quad (13) \text{ 被乘数 } M \\
 \times 1011 \quad (11) \text{ 乘数 } Q \\
 \hline
 1101 \\
 0000 \\
 1101 \\
 1000 \\
 \hline
 10001111 \quad (143) \text{ 乘积 } P
 \end{array}$$

a) 手工相乘算法



b) 阵列实现

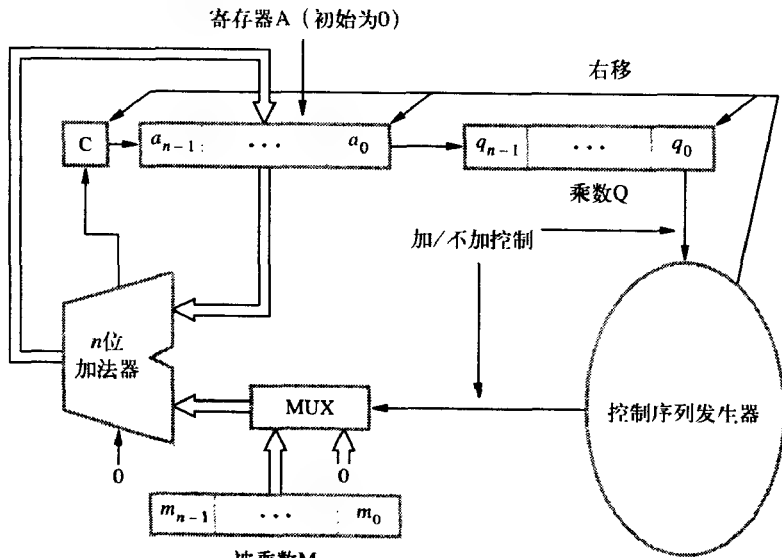
图6-6 二进制正数的阵列乘法

最坏情况下的信号传输延迟路径是从阵列的右上角到阵列左下角的高端乘积位。这条路径包括每行右端的两个单元和最底行的所有单元，呈现出阶梯状模式。假设从全加器部件的输入到输出需要两个门延迟，则对 $n \times n$ 阵列而言，该路径的总延迟为 $6(n-1)-1$ 个门延迟，包括所有单元中初始的与门。（参见习题6.12。）阵列的第一行实际上只需要与门，因为传入的部分积 $PP_0$ 为0。在推导延迟表达式时已经考虑了这一点。

通常在处理器的指令集中提供了乘法操作。高性能处理器芯片使用其上的一部分重要区域执行整数与浮点数的算术运算。（本章的后面会讨论浮点操作。）尽管前面的组合乘法器易于理解，但在计算实际大小的数字（如32位或64位数字）时，它使用了太多的门。乘法操作还可以通过使用图6-6中组合阵列技术与需要较少组合逻辑的时序技术的混合来实现。

进行乘法计算最简单的方法是使用ALU中的加法电路按顺序进行一系列操作。图6-7a的框图显示了顺序乘法电路的硬件组成。该电路通过使用一个 $n$ 位加法器通过 $n$ 次运算来实现图6-6b

中 $n$ 行行波进位加法器执行的空间加法。寄存器A和Q共同容纳 $PP_i$ ，而乘数位 $q_i$ 产生“加/不加”信号。该信号控制被乘数 $M$ 与 $PP_i$ 的加法操作，从而生成 $PP(i+1)$ 。乘积需要 $n$ 个周期计算。从初始向量 $PP_0$ （寄存器A中的 $n$ 个0）开始，在每一周期中部分积 $PP_i$ 的长度都增长一位。加法器的进位存储于触发器C中，它在寄存器A的左侧。开始时，乘数装载到寄存器Q中，被乘数装载到寄存器M中，而C和A被清零。在每一周期的结束，C、A和Q右移一位以容纳部分积的增长，同时乘数移出寄存器Q。由于这种移位，寄存器Q LSB位置上的乘数位 $q_i$ 可以在正确的时刻生成“加/不加”信号，由第一个周期的 $q_0$ 到第二个周期的 $q_1$ 等。这些乘数位使用过后就被右移操作丢弃了。注意，加法器的输出是 $PP(i+1)$ 的最左位，它必须保存在触发器C中，并随A和Q的内容一起右移。 $n$ 个周期之后，乘积的高位部分保存在寄存器A中，而低位部分保存在寄存器Q中。图6-7b显示了以这种硬件安排执行的图6-6a的乘法示例。



a) 寄存器配置

	M					
	1 1 0 1					初始设置
0	0 0 0 0				1 0 1 1	
C	A				Q	
0	1 1 0 1				1 0 1 1	加移位 } 第一个周期
0	0 1 1 0				1 1 0 1	
1	0 0 1 1				1 1 0 1	加移位 } 第二个周期
0	1 0 0 1				1 1 1 0	
0	1 0 0 1				1 1 1 0	不加移位 } 第三个周期
0	0 1 0 0				1 1 1 1	
1	0 0 0 1				1 1 1 1	加移位 } 第四个周期
0	1 0 0 0				1 1 1 1	
乘积						

b) 乘法示例

图6-7 二进制乘法顺序电路

使用这种顺序硬件结构时可以明显地看出, 执行乘法指令所需的时间要比执行加法指令长得多。有几种技术可以加速乘法运算, 我们将在下面的小节中进行讨论。

## 6.4 有符号操作数乘法

我们现在讨论有符号操作数的补码乘法, 它生成长度加倍的积。总的策略依然是加入由乘数位确定的被乘数, 从而累加部分积。

首先, 考虑正乘数和负被乘数的情况。当我们将负的被乘数加到部分积时, 必须将被乘数符号位的值扩展到乘积的最左边。例如在图6-8中, 5位有符号操作数-13为被乘数, 乘以乘数+11, 得到了10位的乘积-143。被乘数的符号扩展如图所示。这样, 如果前面讨论过的硬件提供部分积的符号扩展, 它就可以接受负被乘数。

对于负乘数, 直接的解决办法就是形成乘数与被乘数的补码, 并按照正乘数的情况进行运算。可以这样做是因为对两个操作数求补并不改变乘积的值或符号。下面我们将讨论一种对正负乘数乘法效果同样好的技术, 称为Booth算法。

$$\begin{array}{r}
 \begin{array}{cccccc} & 1 & 0 & 0 & 1 & 1 & (-13) \\ \times & 0 & 1 & 0 & 1 & 1 & (+11) \\ \hline & 1 & 0 & 0 & 1 & 1 & \\ & 1 & 0 & 0 & 1 & 1 & \\ & 0 & 0 & 0 & 0 & 0 & \\ & 1 & 1 & 0 & 0 & 1 & 1 \\ & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & (-143) \end{array}
 \end{array}$$

图6-8 负被乘数的符号扩展

### Booth算法

Booth算法生成 $2n$ 位的乘积, 而且它以统一的方式对待正负补码形式的 $n$ 位操作数。要理解该算法的实质, 考虑一个乘法, 其中乘数为正数而且有惟一的一块区域为全1, 例如0011110。为了得到乘积, 在标准程序中需要将四个适当移位的被乘数相加。但是, 通过将乘数看作是两个数的差可以减少所需操作的数目:

$$\begin{array}{r}
 0100000 \quad (32) \\
 - 0000010 \quad (2) \\
 \hline
 0011110 \quad (30)
 \end{array}$$

这意味着将被乘数的 $2^5$ 倍与 $2^1$ 倍的补码相加即可得到乘积。为了方便, 我们可以将前面的乘数重新编码为 $0 + 1000 - 10$ , 并以此描述所需操作的顺序。

一般而言, 在Booth算法中, 从右至左扫描乘数, 由0变为1时将移位的被乘数乘以-1, 由1变为0时将移位的被乘数乘以+1。图6-9演示了刚才例子的普通算法与Booth算法。很明显, Booth算法可以扩展到包含任意块数1的乘数, 包括将单独的1视为一块的情况。图6-10是对乘数进行重编码的另一个例子。当乘数的最低有效位为1时, 假设其右方隐含包括一个0。Booth算法还可以直接应用于负乘数的情况, 如图6-11所示。

为了证明Booth算法对负乘数的正确性, 我们使用补码系统中负数表示的以下性质: 假设负数 $X$ 中最左方的0在位置 $k$ 处, 即

$$X = 11 \cdots 10x_{k-1} \cdots x_0$$

则 $X$ 的值为

$$V(X) = -2^{k+1} + x_{k-1} \times 2^{k-1} + \dots + x_0 \times 2^0$$

				0	1	0	1	1	0	1
				0	0	+1	+1	+1	+1	0
				0	0	0	0	0	0	0
			0	1	0	1	1	0	1	
		0	1	0	1	1	0	1		
	0	1	0	1	1	0	1			
0	1	0	1	1	0	1				
0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0			
0	0	0	1	0	1	0	1	0	0	1

				0	1	0	1	1	0	1
				0	+1	0	0	0	-1	0
0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	1	0	0	1	1		
0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	
0	0	0	1	0	1	1	0	1		
0	0	0	0	0	0	0	0	0		
0	0	0	1	0	1	0	1	0	0	1

被乘数的补码

图6-9 普通乘法与Booth乘法方案

0	0	1	0	1	1	0	0	1	1	1	0	1	0	1	1	0	0
↓ ↓																	
0	+1	-1	+1	0	-1	0	+1	0	0	-1	+1	-1	+1	0	-1	0	0

图6-10 乘数的Booth重编码

0	1	1	0	1	(+13)			0	1	1	0	1
×	1	1	0	1	0	(-6)		0	-1	+1	-1	0
								0	0	0	0	0
								1	0	0	1	1
								0	1	1	0	1
								1	0	0	1	1
								0	0	0	0	0
								1	1	1	0	1
								1	1	0	1	0
								1	1	0	1	0

(-78)

图6-11 负乘数的Booth乘法

$V(X)$ 表达式的正确性可以这样证明, 即将 $X$ 看作是以下两个数之和:

$$\begin{array}{r} 11 \cdots 100000 \cdots 0 \\ + \quad 00 \cdots 00x_{k-1} \cdots x_0 \\ \hline X = 11 \cdots 10x_{k-1} \cdots x_0 \end{array}$$

其中上面的数是  $-2^{k+1}$  的补码表示。现在, 重编码的乘数的一部分就是和式中的第二个数, 并且其  $k+1$  位为  $-1$ 。例如, 乘数  $110110$  可以重编码为  $0-1+10-10$ 。

图6-12总结了对乘数重编码的Booth技术。由  $011 \cdots 110$  至  $+100 \cdots 0-10$  的转换称作跳过1位。



这个术语是从乘数中包含连续全1的块区域的情况得来的。只需几个移位被乘数（求和项）的版本相加而得到乘积，因此加速了乘法运算。但是，在最坏情况下，即乘数中的1和0交替时，乘数的每一位都将选择一个求和项。实际上，它产生的求和项比不使用Booth算法时还要多。图6-13显示了16位的最坏情况、普通情况和较好情况下的乘数。

Booth算法有两个吸引人的特征。第一，它以统一的方式处理正负乘数；第二，当乘数中包含很多大的全1块区域时，它的效率很高。通过跳过1位而提升的速度依赖于数据。平均情况下，使用Booth算法进行乘法运算的速度与使用普通算法相同。

乘数		第 <i>i</i> 位选择的被乘数版本
第 <i>i</i> 位	第 <i>i</i> -1位	
0	0	$0 \times M$
0	1	$+1 \times M$
1	0	$-1 \times M$
1	1	$0 \times M$

图6-12 Booth乘数重编码表

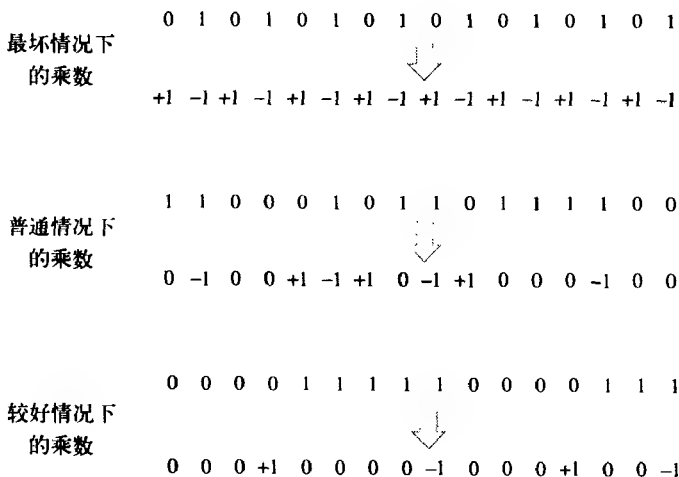


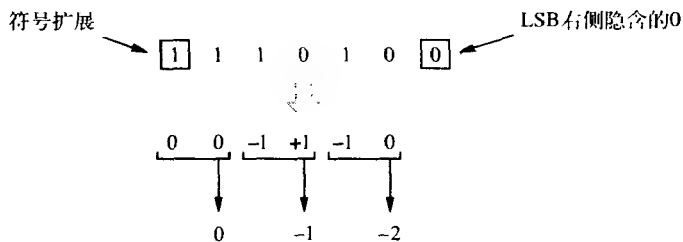
图6-13 Booth重编码的乘数

## 6.5 快速乘法

**383** 现在我们讲述两种提高乘法运算速度的技术。第一种技术保证对于*n*位操作数，参与加法的求和项（被乘数的不同版本）的最大数目为*n*/2。第二种技术减少了计算求和项加法的时间。

### 6.5.1 乘数位偶重编码

位偶重编码（bit-pair recoding）技术可将求和项的最大数目减少一半。它直接派生于Booth算法。将Booth算法重编码的乘数位组成对时，可以观察到：对 $(+1 \ -1)$ 与对 $(0 \ +1)$ 是等价的。也就是说，不是将移位位置*i*上-1倍的被乘数*M*与位置*i* + 1上+1倍的*M*相加，而是在位置*i*加入 $+1 \times M$ ，也可以得到相同的结果。其他的例子有： $(+1 \ 0)$ 与 $(0 \ +2)$ 等价， $(-1 \ +1)$ 与 $(0 \ -1)$ 等价，等等。这样，如果从右边开始对Booth算法重编码的乘数每次检查两位，那么对于每一对乘数位最多只需要加入一个被乘数。图6-14a显示了对图6-11中乘数的位偶重编码，而图6-14b显示了在所有可能的情况下被乘数的选择决策表。图6-15使用乘数的位偶重编码重新计算了图6-11中的乘法。



a) 由Booth重编码派生的位偶重编码

乘数位偶		右侧乘数位	位置 <i>i</i> 选择的被乘数
<i>i</i> + 1	<i>i</i>	<i>i</i> - 1	
0	0	0	0 × M
0	0	1	+1 × M
0	1	0	+1 × M
0	1	1	+2 × M
1	0	0	-2 × M
1	0	1	-1 × M
1	1	0	-1 × M
1	1	1	0 × M

b) 被乘数选择决策表

图6-14 乘数位偶对重编码

## 6.5.2 求和项的进位保留加法

在乘法中需要将几个求和项相加。一种称为进位保留加法 (carry-save addition, CSA) 的技术可以提高加法速度。考虑图 6-16a 中的  $4 \times 4$  乘法阵列。这个结构是图 6-6 中的普通阵列，它的第一行只包含实现位乘积  $m_3q_0$ 、 $m_2q_0$ 、 $m_1q_0$  和  $m_0q_0$  的与门。

我们可以不让进位在各行中波动，而是将它们“保存”起来，并在正确的权重位置上将它们引入到下一行中，如图 6-16b 所示。这释放了第一行中三个全加器的一个输入端。这些输入端用来引入第三个求和项的位乘积  $m_2q_2$ 、 $m_1q_2$ 、 $m_0q_2$ 。现在，第二行中每个全加器的两个输入端装载了来自第一行的和与进位输出。第三个输入端用来引入第四个求和项的位乘积  $m_2q_3$ 、 $m_1q_3$ 、 $m_0q_3$ 。第三和第四个求和项的高端位乘积  $m_3q_2$  与  $m_3q_3$  被引入到第二与第三行左侧其余空闲的输入端中。来自第二行的保留进位位及求和位现在加载到第三行，从而得到最终的乘积位。

进位保留阵列的延迟比行波进位阵列要小一些。这是因为每一行的输出向量  $S$  和  $C$  是在一个全加器延迟中并行生成的。延迟的减少量将在习题 6.22 中考虑。

$$\begin{array}{r} 01101 \quad (+13) \\ \times 11010 \quad (-6) \\ \hline \end{array}$$



$$\begin{array}{r} 01101 \\ 0-1+1-10 \\ \hline 0000000 \\ 110011 \\ 00001101 \\ 1110011 \\ 000000 \\ \hline 1110110010 \quad (-78) \end{array}$$

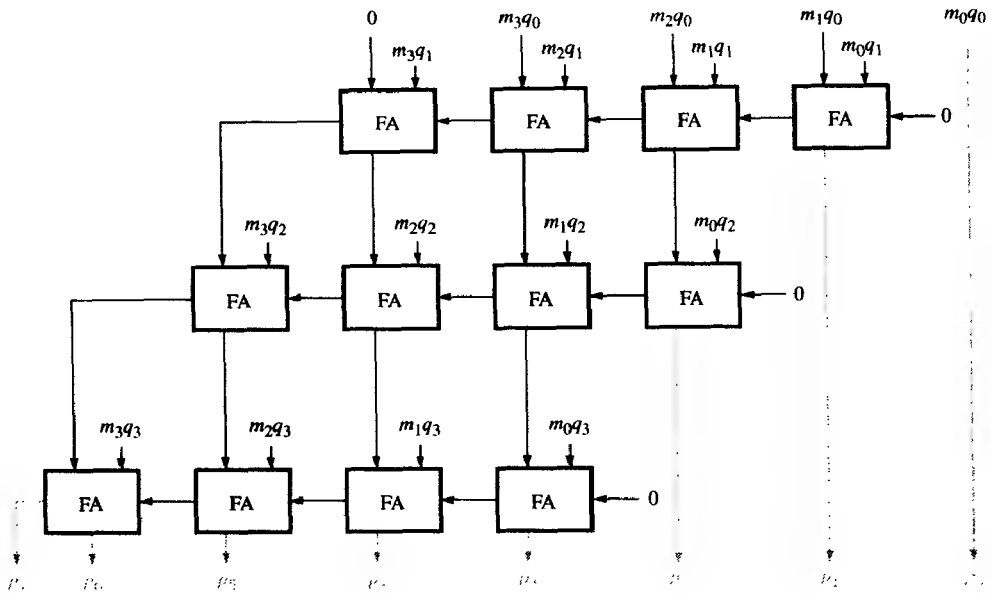


$$\begin{array}{r} 01101 \\ 0-1-2 \\ \hline 1100110 \\ 110011 \\ 000000 \\ \hline 1110110010 \end{array}$$

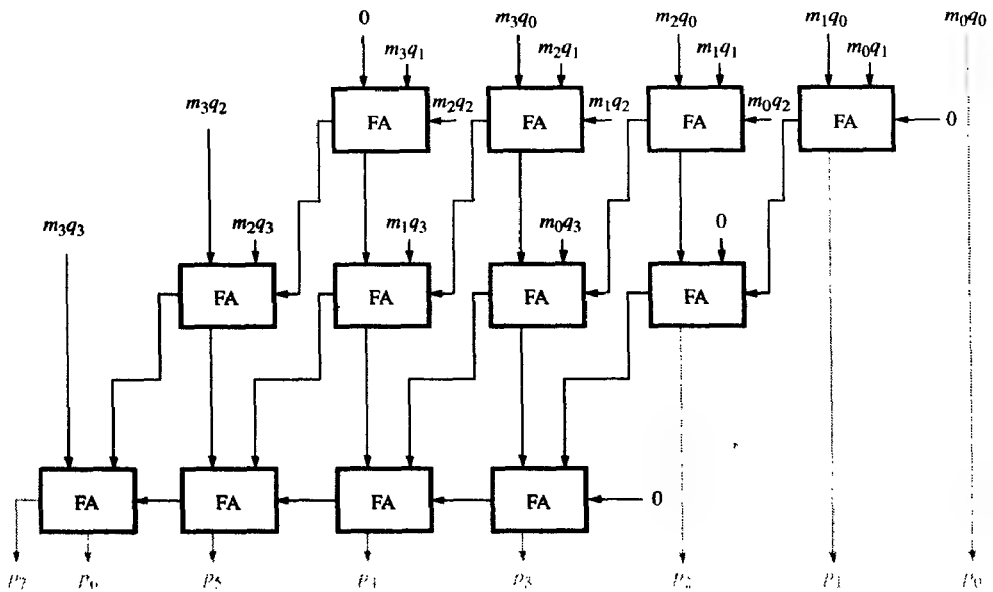
图6-15 乘法中只需  $n/2$  个求和项

384

385



a) 行波进位阵列 (图6-6的结构)



b) 进位保留阵列

图6-16 4位操作数乘法操作 $M \times Q = P$ 的行波进位与进位保留阵列

使用下面的方法可以更显著地减小延迟。在长操作数乘法中需要对许多求和项求和，我们可以将三个求和项分为一组，并对每个组并行地执行进位保留加法，从而在一个全加器延迟中得到一系列 $S$ 与 $C$ 向量。然后将所有的 $S$ 和 $C$ 向量分为三个一组，对它们执行进位保留加法，从而在一个全加器延迟中生成更深一层的 $S$ 与 $C$ 向量。我们将这个过程继续，直到只剩下两个向量。然后它们就可以使用行波进位加法器或超前进位加法器相加，求得最终的乘积。

考虑两个6位无符号数相乘且乘数的全部位均为1时，对六个被乘数的移位版本求和的情况。

图6-17显示了这个例子。图6-18使用进位保留加法对六个求和项 $A, B, \dots, F$ 求和。两幅图中的“黑框”指示了相同的操作数位，还显示了它们是怎样在图6-18的进位保留加法中变为一系列求和位与进位位的。示例中执行了三个层次的进位保留加法，它们在图6-19中表示了出来。从图中可以明显看出，最终的两个向量 $S_4$ 与 $C_4$ 在六个输入求和项加载到第一层三个全加器延迟后得到。生成最终乘积的 $S_4$ 与 $C_4$ 之间的普通加法操作可以使用行波进位加法器或超前进位加法器完成。

387

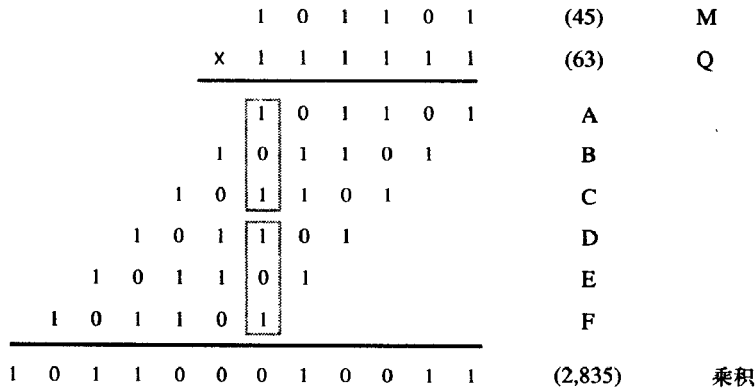


图6-17 用来展示图6-18中进位保留加法的乘法示例

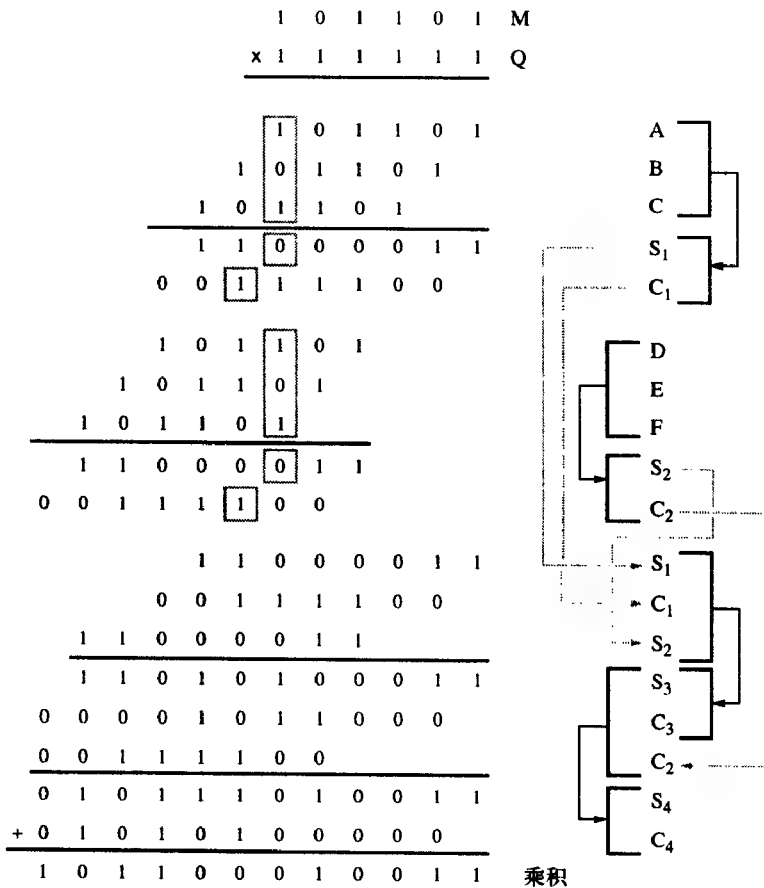


图6-18 使用进位保留加法计算图6-17中的乘法示例

让我们计算一下图6-18和图6-19中进行 $6 \times 6$ 乘法总的逻辑门延迟。在经过根据乘数位选择求和项的一个与门延迟后，所有的六个求和项都加载到第一层CSA。假设每一层CSA有两个门延迟，则第三层CSA的输出 $S_4$ 和 $C_4$ 在六个门延迟后得到。最终两个向量的加法可以使用图6-5的超前进位加法器在八个门延迟后完成。因此总的门延迟为15。作为比较，使用图6-6中的 $n \times n$ 阵列进行乘法运算的总的门延迟为 $6(n-1)-1$ ；所以 $6 \times 6$ 阵列的总门延迟为29。延迟的减半是对求和项并行使用进位保留加法和对两个最终向量使用超前进位加法的共同结果。

388

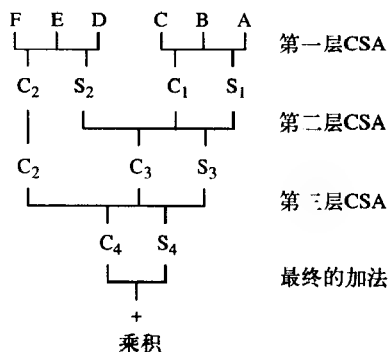


图6-19 图6-18中进位保留加法操作的图示

当求和项的数目很大时，节省的时间也相应地大幅增加。例如，按照图6-19的模式使用进位保留加法对32个数求和，在最终的加法前只需要八层CSA操作。一般而言，可以证明将 $k$ 个求和项降为最终求和的两个向量大约需要 $1.7\log_2 k - 1.7$ 层的CSA操作（见习题6.23）。可以使用64位超前进位加法器计算两个最终向量的加法。 $32 \times 32$ 乘法的总延迟计算如下：初始的与门需要一个门延迟，其输出为32个求和项；八层CSA操作需要16个门延迟；64位加法器的最长路径（到 $s_{63}$ ）需要12个门延迟。因此总延迟为29个门延迟。作为比较，使用阵列乘法器计算 $32 \times 32$ 乘法需要185个门延迟才能得到最后的乘积位。

我们在讨论使用进位保留加法加速乘法操作时忽略了一些问题。首先，当求和项为负时，就像在使用Booth算法计算有符号数乘法时一样，必须在CSA逻辑中加入符号扩展。在这里并不需要将符号扩展到整个两倍的乘积长度，只需在每一层CSA作几位的扩展。其次，我们以前假定对 $n \times n$ 乘法对两个最终向量 $S$ 和 $C$ 求和需要使用 $2n$ 位的超前进位加法器。然而实际上参加最终加法的位数要少，因为一些低位的乘积位已经在早些时候确定了。但这并不影响大局；而且我们曾经使用的延迟分析也是正确的，因为加法器的位数并没有显著缩短；最后，我们对 $n \times n$ 乘法使用了 $n$ 个求和项。但如果使用了乘数的位偶重编码技术，求和项的数目就会降为 $n/2$ 。这也将CSA的层数从 $1.7\log_2 n - 1.7$ 减少为 $1.7\log_2 n - 3.4$ 。

389

### 快速乘法总结

现在我们总结一下快速乘法技术。由Booth算法发展而来的乘数位偶重编码技术可将求和项的数目减少为原来的 $1/2$ 。使用较少数目的进位保留加法可以进一步将求和项的数目降为2。最终的乘积可以通过使用超前进位加法器进行一次加法得到。高性能处理器的设计师已经使用了所有这三种技术——乘数的位偶重编码、求和项的进位保留加法和超前进位加法——来减少执行乘法操作所需的时间。

## 6.6 整数除法

在6.4节讨论整数乘法时，我们将手工和逻辑电路计算乘法操作的方法联系了起来。在这里将使用同样的方法讨论整数除法。我们将详细地讨论正数除法，然后再对有符号操作数的情况作一般性的说明。

图6-20显示了相同数值十进制与二进制除法的示例。先来看一下十进制的情况。商中的2是由以下推理确定的：首先，试着用13去除2，但这不行。然后，试着用13去除27。我们试探着将

13乘以2得到26, 而且 $27 - 26 = 1$ 小于13, 所以上商2并进行所需的减法。被除数的下一位4被放下来, 最后我们用13去除14, 余数为1。还可以用相同的方式讨论二进制除法, 而且由于商中的各位只能为0或1, 所以二进制的除法运算更简单。

用这种笔算方法实现除法的电路操作如下: 它先将除数与被除数适当地对齐并执行减法。如果余数为0或为正, 则可确定商位为1, 而余数用被除数的下一位扩展, 除数重新定位, 再执行下一次减法。反之, 如果余数为负, 则可确定商位为0, 被除数加上除数以恢复原值, 然后除数重新定位, 执行下一次减法。

$$\begin{array}{r} 21 \\ 13 \overline{) 274} \\ \underline{26} \phantom{0} \\ 14 \\ \underline{13} \\ 1 \end{array} \qquad \begin{array}{r} 10101 \\ 1101 \overline{) 100010010} \\ \underline{1101} \phantom{000} \\ 10000 \\ \underline{1101} \phantom{00} \\ 1110 \\ \underline{1101} \\ 1 \end{array}$$

图6-20 笔算除法示例

390

### 恢复除法

图6-21 显示了实现恢复除法的逻辑电路装置。注意它与图6-7中的乘法电路非常相似。操作开始时,  $n$ 位正除数加载到寄存器M, 而 $n$ 位正被除数加载到寄存器Q。寄存器A被清零。除法结束后,  $n$ 位的商将存放在寄存器Q中, 而余数则存放在寄存器A中。所需的减法操作可以使用补码运算方便地完成。A和M附加位位置在减法过程中存放符号位。下面的算法执行了恢复除法。

执行以下操作 $n$ 次:

1. 将A与Q左移一位。
2. 从A中减去M, 并将结果放回A。
3. 如果A的符号为1, 上商 $q_0$ 为0, 并将M加到A中(即恢复A); 否则上商 $q_0$ 为1。

图6-22显示了使用图6-21电路进行4位除法电路的例子。

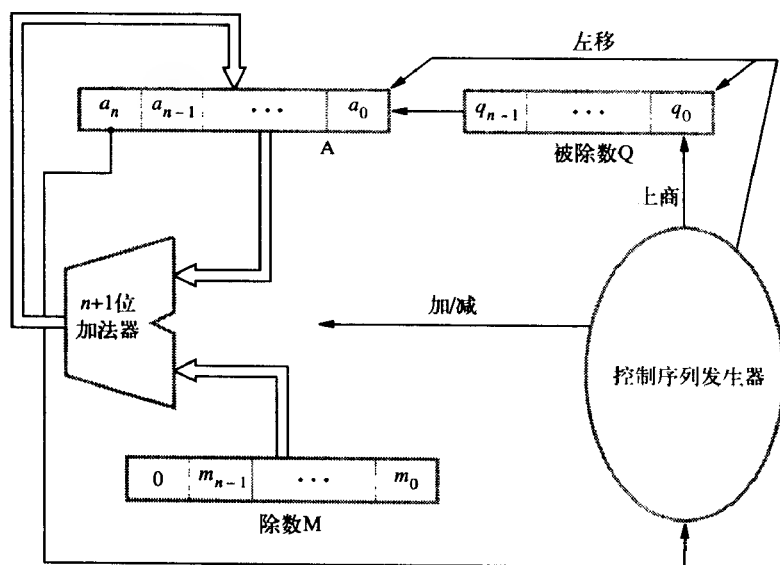


图6-21 二进制除法的电路装置

### 不恢复除法

在不成功的减法之后避免恢复A的值可以提高恢复除法的效率。如果结果为负, 则减法不成功。考虑前面算法中减法操作之后所发生的操作步骤。如果A为正, 左移并减去M, 也即执行 $2A - M$ 。如果A为负, 就执行 $A + M$ 来恢复它, 然后再左移并减去M。这和执行 $2A + M$ 是等价的。正确的操作执行后,  $q_0$ 位就被恰当地置为0或1。我们可以用下面的算法对不恢复除法作一个总结。

391

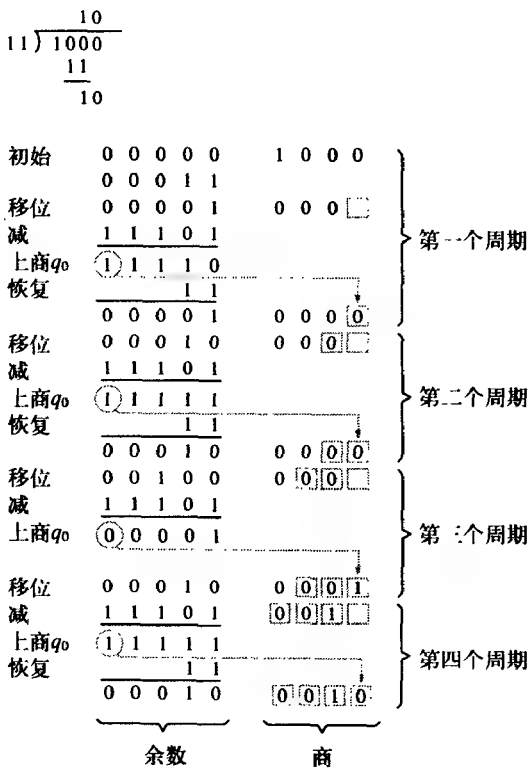
第1步: 执行以下步骤 $n$ 次。

1. 如果A的符号为0, 将A和Q左移一位, 并用A减去M; 否则, 将A和Q左移, 并将M加到A上。

2. 现在, 如果A的符号为0, 上商 $q_0$ 为1; 否则, 上商 $q_0$ 为0。

第2步: 如果A的符号为1, 将M加到A上。

392 在执行第1步 $n$ 次之后, 需要执行第2步以便在A中设置适当的正余数。图6-21的逻辑电路也可以执行这个算法。注意, 这里已经不再需要恢复的操作, 而且每一轮只执行一次加法或减法操作。图6-23显示了使用不恢复除法计算图6-22中示例的情况。



$$F(B) = -b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-(n-1)} \times 2^{-(n-1)}$$

$F$ 的范围如下

$$-1 < F < 1 - 2^{-(n-1)}$$

考虑以32位有符号定点格式表示的数的取值范围。解释为整数时，取值范围大约为0到 $\pm 2.15 \times 10^9$ 。如果我们将它看作小数，取值范围大约为 $\pm 4.55 \times 10^{-10}$ 到 $\pm 1$ 。在科学计算中我们可能会接触到一些参数，例如阿伏加德罗（Avogadro's）常数（ $6.0247 \times 10^{23}$  摩尔 $^{-1}$ ）或者普朗克（Planck's）常量（ $6.6254 \times 10^{-27}$  erg·s），因此上面的两种取值范围都是不够的。我们需要容易地将非常大的整数和非常小的小数包含进来。要做到这一点，二进制小数点的位置应该是可变的并且随着计算的进行自动调整，而计算机必须能够以这种形式表示数字及其操作。在这种情况下，我们说二进制小数点是浮动的，并称数字为浮点数。而定点数的二进制小数点的位置是不变的，这是它们的主要区别。

因为浮点数中二进制小数点的位置是可以变化的，所以在浮点表示中必须明确指出小数点的位置。例如，在我们熟悉的十进制科学计数法中，数字可以记为 $6.0247 \times 10^{23}$ ， $6.6254 \times 10^{-27}$ ， $-1.0341 \times 10^2$ ， $-7.3000 \times 10^{-14}$ 等等。我们说这些数具有5位有效数字。它们的比例因子（ $10^{23}$ ， $10^{-27}$ 等）指示了相对于有效数字的十进制小数点的位置。习惯上，当十进制小数点处于第一个（非零）有效数字右方时，我们说这个数是规格化的。注意比例因子中的基数10是固定的，并不需要在浮点数的机器表示中明确出现。符号、有效数字和比例因子中的指数构成了浮点数的表示。因此我们可以将浮点数的表示定义为：用符号、一串有效数字（通常称作尾数）以及对应于比例因子中隐含基数的指数来表示一个数的方法。

### 6.7.1 浮点数的IEEE 标准

我们从十进制系统浮点数的一般格式和大小出发，引出相应的二进制表示。一个有用的格式如下：

$$\pm X_1 X_2 X_3 X_4 X_5 X_6 X_7 \times 10^{\pm Y_1 Y_2}$$

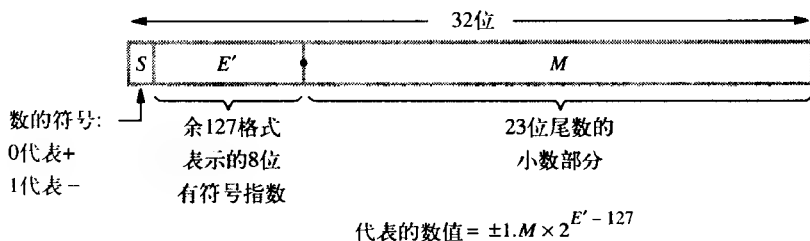
其中 $X_i$ 与 $Y_i$ 为十进制数。在这个表示中，有效数字的个数（7）和指数的范围（ $\pm 99$ ）对于大量的科学计算已经足够了。32位（标准计算机字长）的二进制表示可以接近这个尾数精度和比例因子范围。24位的尾数可以大致表示一个7位十进制数，而基数为2的8位指数所提供的比例因子范围也比较合适。表示数的符号需要一个二进制位。因为规格化二进制尾数的前导非零位必定为1，所以这一位不需要在表示中出现。因此，总共需要32位。

这个用32位表示浮点数的标准已经由电气和电子工程师协会（IEEE）<sup>[1]</sup>制定并详细说明。该标准既描述了表示方法，也描述了四种基本算术运算的执行方式。图6-24a给出了浮点数的32位表示。第一位表示数的符号，接着是比例因子（以2为基数）的指数表示方式。指数域中实际存储的并不是有符号的指数 $E$ ，而是一个无符号整数 $E' = E + 127$ 。这称作余127格式。这样 $E'$ 的取值范围为 $0 < E' < 255$ 。这个范围的端点值0和255用来表示特殊值，我们在后面会讲到。因此 $E'$ 的正常取值范围为 $1 < E' < 254$ 。这意味着实际指数 $E$ 的取值范围为 $-126 < E < 127$ 。指数的余 $x$ 表示使我们可以高效地比较两个浮点数的相对大小。（参见习题6.27。）

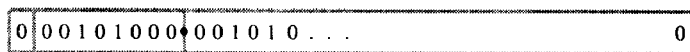
最后的23位代表尾数。因为使用了二进制规格化，尾数的最高有效位恒为1。这一位没有明确地表示出来；我们假定它在紧靠在二进制小数点的左方。这样， $M$ 域中存储的23位实际上



表示了尾数的小数部分,也即二进制小数点右方的各位。图6-24b中显示了一个单精度浮点数的例子。

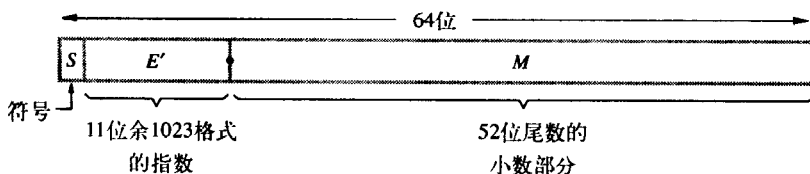


a) 单精度



代表的数值 =  $1.001010 \dots 0 \times 2^{-87}$

b) 单精度数示例



代表的数值 =  $\pm 1.M \times 2^{E' - 1023}$

c) 双精度

图6-24 IEEE标准浮点格式

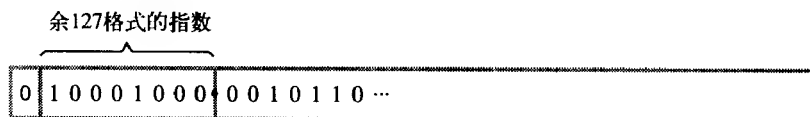
图6-24a中的32位标准表示称为单精度格式,因为它占用了一个32位的字。比例因子的范围为 $2^{-126}$ 到 $2^{+127}$ ,这大致接近 $10^{\pm 38}$ 。24位尾数所提供的精度与7位十进制值相仿。为了提高浮点数的精度与取值范围,IEEE标准还指定了双精度格式,如图6-24c所示。双精度格式增大了指数与尾数的取值范围。11位余1023指数 $E'$ 的正常取值范围为 $1 < E' < 2046$ ,0和2047用来指示特殊值。这样,实际指数 $E$ 的范围为 $-1022 < E < 1023$ ,它所提供的比例因子为从 $2^{-1022}$ 到 $2^{1023}$ (接近 $10^{\pm 308}$ )。53位尾数所提供的精度与16位十进制数相当。

计算机必须至少提供单精度表示以符合IEEE标准。双精度表示是可选的。标准还规定了这两种表示的几种可选的扩展版本。这些扩展版本的目的是为计算过程中的中间值表示提供更高的精度和更大的指数范围。例如,两个向量的点乘积可以通过累加扩展精度的乘积和得到。输入以标准精度表示,即单精度或双精度,计算结果也截取为同样的精度。使用扩展格式有助于减小计算过程中累加的舍入误差。扩展格式还可以提高基本函数如正弦、余弦等的精确度。除了四种基本的算术操作,标准还要求提供余数、平方根以及二进制与十进制表示的转换等操作。

请注意浮点数操作的两个基本方面。第一,如果数字没有规格化,那么总可以通过移位小数和调整指数将它转化为规格化格式。图6-25显示了一个非规格化数 $0.0010110 \dots \times 2^9$ ,以及它的规格化形式 $1.0110 \dots \times 2^6$ 。因为比例因子的形式为 $2^i$ ,将尾数向左或向右移动一位可以分别通

过对指数加1或减1来补偿。第二, 计算过程中可能会产生正常数字表示范围之外的数字。对于单精度, 这意味着该数的规格化表示中的指数需要小于-126或者大于+127。在第一种情况中, 我们说发生了下溢, 而在第二种情况中, 我们说发生了溢出。下溢与溢出都属于我们将要讨论的算术异常。

396



(二进制小数点左方没有隐含的1)

代表的值 =  $+0.0010110... \times 2^9$

a) 未规格化值



代表的值 =  $+1.0110... \times 2^6$

b) 规格化版本

图6-25 IEEE单精度格式浮点数规格化

### 特殊值

余127指数 $E'$ 的端点值0和255被用来表示特殊值。当 $E' = 0$ 且尾数的小数部分 $M$ 也为0时, 表示的是数值0。当 $E' = 255$ 且 $M = 0$ 时, 表示的是值 $\infty$ , 这里 $\infty$ 是用0去除正常数字的结果。符号位仍然是表示的一部分, 所以存在 $\pm 0$ 和 $\pm \infty$ 的表示。

当 $E' = 0$ 且 $M \neq 0$ 时, 表示了非规格化数。其值为 $\pm 0.M \times 2^{-126}$ 。因此它们比最小的规格化数还要小。这时在二进制小数点的左方没有隐含的1, 而 $M$ 是任何非零的23位小数。引入非规格化数的目的是实现逐级下溢, 为某些处理极小数值的情况提供对可以正常表示的数值范围的扩展。当 $E' = 255$ 且 $M \neq 0$ 时, 表示的是非数(或无定义数, Not a Number, NaN)。NaN是执行非法操作的结果, 例如:  $0/0$ 或 $\sqrt{-1}$ 。

### 异常

为了符合IEEE标准, 执行操作时如果发生以下事件, 处理器必须设置异常标志: 下溢、溢出、除0、不精确和非法。我们已经讲过前三种事件。不精确描述的是数字需要进行舍入才能用规范格式表示的情况。非法异常在执行 $0/0$ 或 $\sqrt{-1}$ 操作时发生。当异常发生时, 计算结果就被设为特殊值。

397

如果允许对某个异常标志中断, 当异常发生时就会进入系统或用户定义的例程。或者, 如果必要, 程序可以检测异常的发生, 并决定接下来需要做什么。

对本节与下面两小节浮点问题更详细的讨论见附录A中的Hennessy和Patterson<sup>[2]</sup>。

## 6.7.2 浮点数算术运算

本节我们将概述浮点数的加、减、乘、除运算的基本步骤。所给出的规则适用于单精度IEEE标准格式。这些规则只说明了执行四则运算的主要步骤; 例如, 我们没有讨论可能发生的

溢出或下溢。还有,尾数与指数的中间结果需要的位数可能比24和8位更多。在设计符合标准的算术部件时,必须认真考虑操作中的这些问题。尽管在规则说明中没有给出全部的细节,我们还是考虑了实现中的一些方面,包括后面会讨论的舍入问题。

在进行加减运算之前,如果浮点数的尾数不同,则尾数必须根据对方进行移位操作。考虑一个十进制的例子,将 $2.9400 \times 10^2$ 与 $4.3100 \times 10^4$ 相加。我们将 $2.9400 \times 10^2$ 重写为 $0.0294 \times 10^4$ ,再进行尾数加法从而得到 $4.3394 \times 10^4$ 。加减法的规则如下:

#### 加/减法规则

1. 选取指数较小的数字,将其尾数右移,右移的步数等于两指数之差。
2. 将结果的指数设为与较大的指数相等。
3. 对尾数进行加/减运算,并确定结果的符号。
4. 如果必要,则对结果的值进行规格化。

乘法比加减法要简单些,因为它们不需要对齐尾数。

#### 乘法规则

1. 将指数相加并减去127。
2. 将尾数相乘并确定结果的符号。
3. 如果必要,则对结果的值进行规格化。

#### 除法规则

1. 将指数相减并加上127。
2. 将尾数相除并确定结果的符号。
3. 如果必要,则对结果的值进行规格化。

在乘法与除法规则中需要加/减127是因为我们对指数使用余127表示。

### 6.7.3 保护位与截取

现在我们来考虑一下实现以上算法时的一些重要方面。尽管初始操作数和最终结果的尾数都限制在24位(包括隐含的前导1),在中间步骤中保留一些附加位,通常称为保护位,是非常重要的。它们使结果具有最高的精确度。

生成最终结果时去除保护位要求对扩展的尾数进行截取,从而获得接近扩展尾数的24位数字。在其他一些情况下也会进行这一操作,比如将十进制数转换成二进制时。应当指明舍入也可以用于描述截取操作,但是在这里我们将舍入更严格地定义为一种特定格式的截取。

截取有几种方法。最简单的方法是将保护位去除,而对剩余的各位不加改变。这种方式称作截断。假设我们要使用这种方法将一个小数从六位截取到三位。区间 $0.b_{-1}b_{-2}b_{-3}000$ 到 $0.b_{-1}b_{-2}b_{-3}111$ 内的所有小数都将截取为 $0.b_{-1}b_{-2}b_{-3}$ 。三位结果的误差为从0到0.000111。换句话说,截断误差为从0到接近剩余位最低有效位置上的1。在上面的例子中,该位置就是 $b_{-3}$ 的位置。截断的结果是有偏近似,因为误差区间并不关于0对称。

另一种最简单的截取方法是冯·诺依曼舍入。如果要去掉的各位全为0,就简单地去除它们,并保持其他位不变。但是如果要去掉的任何一位为1,就将剩余各位的最低有效位设为1。在六位到三位的截取例子中,所有 $b_{-4}b_{-5}b_{-6}$ 不等于000的小数都被截取为 $0.b_{-1}b_{-2}1$ 。这种截取方法的误差在剩余位LSB位置的-1到+1之间。尽管这种技术的误差比截断要大,但误差的最大值相同,

而且其近似是无偏的, 因为误差区间关于0对称。

如果在生成结果时涉及许多操作数和操作, 那么无偏近似是有利的, 因为随着计算的进行正误差会与负误差相互抵消。在统计上, 复杂计算的结果很可能具有较高的精确度。

第三种截取方法是舍入过程。舍入的结果最接近于被截取的数字, 而且是无偏近似。其过程如下: 如果去除位的MSB位置上为1, 则在保留位的LSB位置上加1。这样,  $0.b_{-1}b_{-2}b_{-3}1\cdots$ 就被舍入成 $0.b_{-1}b_{-2}b_{-3} + 0.001$ , 而 $0.b_{-1}b_{-2}b_{-3}0\cdots$ 则被舍入为 $0.b_{-1}b_{-2}b_{-3}$ 。这种方法提供了我们需要的近似, 除了去除位为 $10\cdots0$ 的情况。这是一个中间状态; 待截取的数字位于两个最近的截取表示的正中间。为了无偏地解决问题, 一种可能是让保留位设置为最接近的偶数。对于六位例子, 数值 $0.b_{-1}b_{-2}0100$ 将被截取为 $0.b_{-1}b_{-2}0$ , 而数值 $0.b_{-1}b_{-2}1100$ 则被截取为 $0.b_{-1}b_{-2}1 + 0.001$ 。“在陷入僵局时舍入到最近的数或最近的偶数”这句话有时就是描述这种截取技术。误差区间大致在保留位LSB位置的 $-1/2$ 到 $+1/2$ 。很明显, 这是最好的方法。但是, 它也是最难实现的, 因为它要求一次加法操作以及可能的再规格化。IEEE浮点标准将这种舍入技术指定为截取操作的默认模式。标准规定了其他的截取方法, 并将所有这些方法称为舍入模式。

399

这里对通过截取去除保护位引入误差的讨论只考虑了单独的截取操作。当对浮点数进行一系列很长的计算时, 确定最终结果误差区间或范围的分析将会变成复杂的研究。除了对IEEE浮点标准中保护位和舍入的处理方式作一些讨论之外, 我们将不再进一步讨论数值计算等方面。

单步操作的结果必须精确于LSB位置上单位的一半。一般情况下, 这要求采用舍入作为截取方法。实现舍入方法只需要在执行操作的中间步骤中添加三个保护位。其中前两位是将被去除的尾数部分中的两个最高有效位。第三位是在尾数的完整表示中以上两位之后所有各位的逻辑或。这一位在执行操作的中间步骤中比较容易维护。它应当初始化为0。如果一个1从该位置移出, 则将这一位设为1, 并保持该值; 因此, 它通常被称为粘着位。

#### 6.7.4 浮点操作的实现

浮点操作的硬件实现涉及大量的逻辑电路。这些操作也可以用软件例程实现。不论使用哪种方式, 计算机必须能够将用户十进制表示的数字转换为所需要的输入格式, 并将输出转换为十进制表示。大多数通用处理器在机器指令级提供浮点操作, 并用硬件实现。

图6-26显示了一个浮点操作实现的例子。这是具有图6-24a格式的32位浮点操作数加减操作硬件实现的结构图。按照6.7.2节中的加/减法规则, 我们看到第1步是比较指数以确定对具有较小指数数字的尾数进行移位的次数。移位次数 $n$ 由图中左上角的8位减法器电路决定。 $E'_A - E'_B$ 之差的值 $n$ 被传送到SHIFTER (移位器) 部件。如果 $n$ 大于操作数的有效位数目, 则结果实际上就是较大的操作数 (不考虑舍入中的保护位与粘着位), 而且可以采用便捷的方法产生结果。对此我们不进行详细讨论。

400

比较指数所得的差的符号决定对哪个数的尾数移位。因此在第1步中, 这个符号被传入图6.26右上角的SWAP (交换器) 网络。如果符号为0, 则 $E'_A > E'_B$ , 尾数 $M_A$ 与 $M_B$ 直接通过SWAP网络。这使得 $M_B$ 被送入SHIFTER, 并被右移 $n$ 位。另一个尾数 $M_A$ 则被直接送入尾数加法器/减法器。如果符号为1, 则 $E'_A < E'_B$ , 在尾数送至移位器之前需要将它们交换。

第2步由图中靠近左下角的多路复用器MUX完成。结果的指数 $E'$ 临时由第1步中指数比较所得的差的符号确定, 如果 $E'_A > E'_B$ 则 $E' = E'_A$ , 如果 $E'_A < E'_B$ 则 $E' = E'_B$ 。

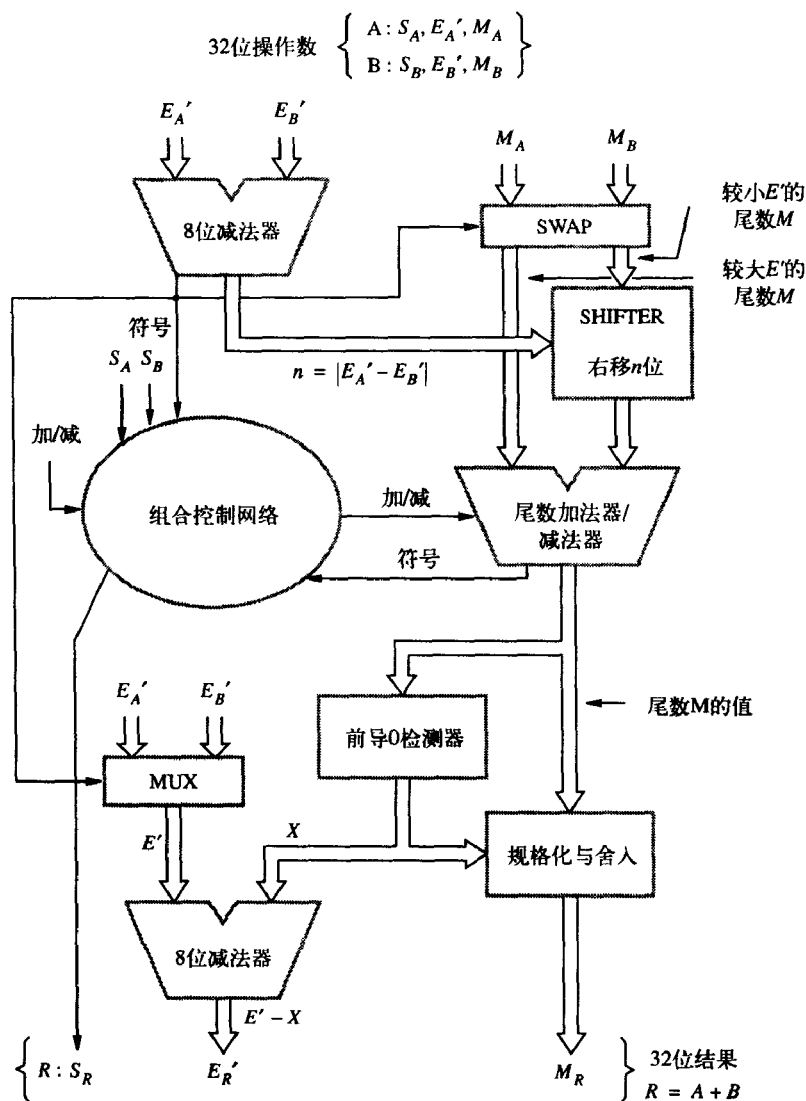


图6-26 浮点加法/减法部件

第3步涉及到了主要组件——图中部的尾数加法器/减法器。CONTROL（控制）逻辑确定尾数是相加还是相减。这由操作数的符号（ $S_A$ 与 $S_B$ ）和操作数上执行的操作（加或减）决定。CONTROL逻辑还决定结果 $S_R$ 的符号。例如，如果A为负（ $S_A=1$ ），B为正（ $S_B=0$ ），且操作为 $A-B$ ，则需要对尾数执行加法且结果的符号为负（ $S_R=1$ ）。反之，如果A和B都为正，且操作为 $A-B$ ，则需要将尾数相减。现在结果的符号 $S_R$ 依赖于尾数的相减操作。例如，如果 $E_A' > E_B'$ ，则 $M_A -$ （移位的 $M_B$ ）为正，因而结果为正。从这个例子可以看出，指数比较所得的符号也需要作为CONTROL网络的输入。当 $E_A' = E_B'$ 且指数相减时，尾数加法器/减法器输出的符号决定了结果的符号。读者现在应该可以为CONTROL网络构造完整的真值表了。

加/减法规则的第4步是对第3步的结果尾数 $M$ 进行规格化。 $M$ 中前导0的数目决定了对 $M$ 的移位次数 $X$ 。规格化的值被截取从而生成24位的结果尾数 $M_R$ 。临时结果的指数 $E'$ 也需要减去值 $X$ ，

生成真正的结果指数 $E'_R$ 。注意,可能只需要右移一位就可以对结果规格化。两个形式为 $1.xx\cdots$ 的尾数相加时就会发生这种情况。这时 $M$ 向量具有 $1x.xx\cdots$ 的形式,这对应于图中 $X$ 的值为 $-1$ 的情况。

对于中间尾数值必须附带的保护位,我们没有给出任何细节。在IEEE标准中,如前所述,生成结果中的24位规格化尾数只需要几个保护位即可。

让我们考虑一下实现6-26结构图所需要的实际硬件。如前所述,两个8位减法器 and 尾数加法器/减法器可以使用组合逻辑实现。因为它们必须以原码的形式输出,因此对前面的讨论做一些修改。经常需要使用反码运算与原码表示的组合。SHIFTER和输出规格化操作的实现允许有极大的灵活性。如果设计需要使逻辑门数达到最小,那么可以使用移位寄存器实现操作。然而,为了达到高性能,也可以将它们构造为组合逻辑部件,但是这样会需要大量的逻辑门。在高性能处理器中,相当数量的芯片区域是用于浮点操作的。

402

## 6.8 结束语

计算机的算术运算涉及几个非常有趣的逻辑设计问题。本章讨论了一些在二进制算术部件设计中有实用价值的技术。超前进位技术是高性能加法器设计的主要思想。在快速乘法器设计中,由Booth算法发展而来的乘数位偶重编码减少了生成乘积所需求和项的数目。进位保留加法显著地减少了求和项加法所需的时间。

本章介绍了浮点数的IEEE表示标准,以及执行四则运算的基本规则。为了考察浮点操作实现电路的复杂性,我们讨论了加法/减法部件。

## 习题

6.1 下列加减法问题中的数字为补码表示的6位有符号数。执行题目中指示的操作,说明是否发生了算术溢出,然后将操作数与结果转换为十进制原码的表示形式,以检查答案的正确性。

010110	101011	111111
<u>+001001</u>	<u>+100101</u>	<u>+000111</u>
011001	110111	010101
<u>+010000</u>	<u>+111001</u>	<u>+101011</u>
010110	111110	100001
<u>-011111</u>	<u>-100101</u>	<u>-011101</u>
111111	000111	011010
<u>-000111</u>	<u>-111000</u>	<u>-100010</u>

6.2 6.7小节的开始讨论了补码表示的有符号二进制小数。

- 将二进制数值 $0.5$ ,  $-0.123$ ,  $-0.75$ 和 $-0.1$ 表示为6位小数。(参阅附录E中的十进制到二进制小数转换。)
- 当二进制小数点后使用5位有效数字时,最大表示误差 $e$ 是多少?
- 计算使 $e$ 符合以下条件时二进制小数点后所需的有效数字位数:

403

- (a)  $e < \frac{1}{10}$   
 (b)  $e < \frac{1}{100}$   
 (c)  $e < \frac{1}{1000}$   
 (d)  $e < \frac{1}{10^6}$

6.3 反码与补码二进制表示方法是以 $b$ 为基数的数字系统中 $(b-1)$ 进制补码和 $b$ 进制补码表示技术的特例。例如, 考虑十进制系统。以原码形式表示的数字+526、-526、+70和-70在两个补码系统中都有4位原码表示, 如图P6-1所示。数字的每一位对9取补就构成了九进制补码。对九进制补码加1就构成了十进制补码。在下面的两种表示中, 正数的最左位为0, 负数的最左位为9。

表示	示例			
符号数值	+526	-526	+70	-70
九进制补码	0526	9473	0070	9929
十进制补码	0526	9474	0070	9930

图P6-1 习题6.3中以10为基数的有符号数

现在考虑基数为3的系统(三进制系统), 其中5位无符号数 $t_4t_3t_2t_1t_0$ 的值为 $t_4 \times 3^4 + t_3 \times 3^3 + t_2 \times 3^2 + t_1 \times 3^1 + t_0 \times 3^0$ ,  $0 \leq t_i \leq 2$ 。将原码表示的三进制数+11011、-10222、+2120、-1212、+10和-201表示为三进制补码系统中的6位有符号三进制数。

- 6.4 将十进制数56、-37、122和-123表示为三进制补码格式的6位有符号数, 对所有可能的配对进行加减操作, 并对每个操作说明是否出现了算术溢出。(参见习题6.3对三进制数系统的定义, 并使用类似于附录E中十进制到三进制整数转换的技术。)
- 6.5 半加器(half adder)是一个组合逻辑电路, 它有两个输入 $x$ 与 $y$ , 以及两个输出 $s$ 与 $c$ , 分别是 $x$ 与 $y$ 相加所得的和与进位输出。
- (a) 采用二级与或电路设计一个半加器。
- (b) 说明如何使用两个半加器以及必要的外部逻辑门实现图6-2a中的全加器。
- (c) 比较(b)部分的网络 and 图6-2a中的加法器网络的最长逻辑延迟路径。
- 404 6.6 编写一个68000或IA-32程序, 将16位正二进制数转换为5位BCD码十进制数。BCD数字编码占用内存中五个连续字节的低4位。使用连续除10的转换技术。这个方法类似于附录E中十进制到二进制转换时的连续除2。参考附录C(68000)或D(IA-32)中除法指令的格式与操作。
- 6.7 假设表示0到9999的十进制整数的四个BCD数被装载到32位内存区域DECIMAL的低16位。编写一个ARM、68000或IA-32子程序, 将DECIMAL中存储的十进制整数转换为二进制表示, 并存入内存区域BINARY。
- 6.8 对BCD数求和需要模10加法器。两个BCD数 $A = A_3A_2A_1A_0$ 和 $B = B_3B_2B_1B_0$ 的模10加法执行如

下：将 $A$ 加到 $B$ 上（二进制加法）。之后，如果结果为大于或等于 $10_{10}$ 的非法编码，则加 $6_{10}$ 。（忽略本次加法的溢出。）

(a) 什么时候输出进位等于1？

(b) 证明本算法对于以下数值可以给出正确结果：

$$(1) A = 0101 \quad B = 0110$$

$$(2) A = 0011 \quad B = 0100$$

(c) 使用4位二进制加法器以及必要的外部逻辑门设计一个BCD数加法器。输入为 $A_3A_2A_1A_0$ 、 $B_3B_2B_1B_0$ 与进位输入。输出为和数 $S_3S_2S_1S_0$ 与进位输出。级联这样的组件可以构成行波进位BCD加法器。

6.9 使用适当的真值表证明，在补码整数加法中，逻辑表达式 $c_n \oplus c_{n-1}$ 正确地指示了溢出的发生。

6.10 (a) 使用四个图6-5中16位超前进位加法器以及附加逻辑设计一个64位加法器，从图中的 $c_0$ 、 $G_i''$ 和 $P_i''$ 变量生成 $c_{16}$ 、 $c_{32}$ 、 $c_{48}$ 和 $c_{64}$ 。附加逻辑与图中每个超前进位加法器内部逻辑有什么关系？

(b) 证明6.2.1小节末尾的结论，通过64位加法器的延迟对 $s_{63}$ 为12个门延迟，对 $c_{64}$ 为7个门延迟。

(c) 比较(a)部分中64位加法器与6.2.1小节讨论的使用两个16位加法器级联构成的32位加法器生成 $s_{31}$ 和 $c_{32}$ 的门延迟。

6.11 (a) 构建图6-4中的4位超前进位加法器需要多少逻辑门？

(b) 使用(a)部分的适当结果计算构建图6-5中的16位超前进位加法器需要多少逻辑门。

6.12 证明6.3小节的结论，通过图6-6b中的 $n \times n$ 阵列最坏情况的延迟为 $6(n-1)-1$ 个门延迟。

6.13 用手工方法计算5位无符号数 $A = 10101$ 和 $B = 00101$ 上的操作 $A \times B$ 与 $A \div B$ 。

405

6.14 创建与图6-7b和图6-23类似的图表，说明习题6.13中的乘除法操作是如何使用图6-7a和图6-21的硬件执行的。

6.15 编写ARM、68000或IA-32程序，模仿图6-7中的技术计算两个32位无符号数的乘法。假设乘数与被乘数分别在寄存器 $R_2$ 和 $R_3$ 中。乘积将存入寄存器 $R_1$ （高位部分）和 $R_2$ （低位部分）。（提示：使用移位与循环移位的组合进行双寄存器移位。）

6.16 编写ARM、68000或IA-32程序，基于不恢复除法算法计算整数除法。假设两个操作数均为正数，即被除数和除数的最左位为0。

6.17 使用Booth算法计算以下各对有符号补码数的乘法。假设 $A$ 为被乘数， $B$ 为乘数。

$$(a) A = 010111 \quad B = 110110$$

$$(b) A = 110011 \quad B = 101100$$

$$(c) A = 110101 \quad B = 011011$$

$$(d) A = 001111 \quad B = 001111$$

6.18 使用乘数位偶重编码重复习题6.17。

6.19 概括地说明怎样修改图6-7a中的电路图，使其使用Booth算法实现 $n$ 位补码有符号数乘法。明确指出控制序列发生器（Control sequencer）的输入和输出，以及对加法器和 $A$ 寄存器所作的必要改变。

6.20 如果两个 $n$ 位补码有符号数的乘积可以用 $n$ 位表示，则图6-6a中的手工相乘算法就可以直接



使用, 这时将符号位与其他位同等对待。对下面的两对4位有符号数进行乘法计算:

(a) 乘数 = 1110 被乘数 = 1101

(b) 乘数 = 0010 被乘数 = 1110

为什么这时的计算是正确的?

- 6.21 一个能够计算16位无符号数加法和乘法的整数运算部件可以用来计算两个32位无符号数的乘法。所有的操作数、中间结果和最终结果存储于标号为 $R_0$ 到 $R_{15}$ 的寄存器中。硬件乘法器将 $R_i$  (被乘数) 与 $R_j$  (乘数) 相乘, 并将32位的乘积存入寄存器 $R_j$ 和 $R_{j+1}$ , 其中低位部分存入 $R_j$ 。当 $j = i - 1$ 时, 乘积将覆盖两个操作数。硬件加法器将 $R_i$ 与 $R_j$ 的内容相加并将结果存入 $R_j$ 。加法操作的进位输入为0, 而有进位加法操作的进位输入为进位标志C的值。加法器的进位输出总保存在C中。

说明计算 $R_1$ 、 $R_0$ 和 $R_3$ 、 $R_2$ 中两个32位操作数 (高位部分在前) 乘法的步骤, 将64位的乘积放入寄存器 $R_{15}$ 、 $R_{14}$ 、 $R_{13}$ 和 $R_{12}$ 中。如果需要,  $R_{11}$ 到 $R_4$ 的任何寄存器都可以用来保存中间结果。过程中的每一步可以是乘法、加法或寄存器传输操作。

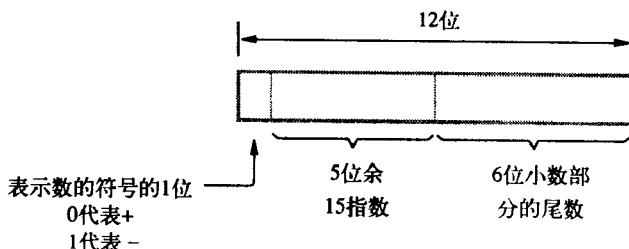
- 6.22 (a) 计算图6-16中每个阵列生成乘积位 $p_7$ 的延迟, 即逻辑门延迟。假设全加器在输入加载两个门延迟后可以得到所有输出。包括开始时生成所有 $m_i q_j$ 乘积的与门延迟。

(b) 6.4小节说明了将图6-16a扩展为 $n \times n$ 阵列的延迟为 $6(n-1)-1$ 。为将图6-16b扩展为 $n \times n$ 阵列生成类似的表达式。

- 6.23 将 $k$ 个求和项减少为两个向量所需的进位保留加法步骤数为 $1.7\log_2 k - 1.7$ , 推导这一公式。(这个公式在6.5.2节给出, 但没有推导。)

- 6.24 (a) 使用与图6-19相似的模式, 将16个求和项降为两个需要多少层CSA操作?  
(b) 画出将32个求和项降为两个的模式图, 以证明6.5.2节八层的结论是正确的。  
(c) 比较(a)与(b)部分的精确答案与从 $1.7\log_2 k - 1.7$ 得出的近似答案。

- 6.25 在6.7节中, 我们使用了实用的32位IEEE标准格式来表示浮点数。这里使用一种缩短格式, 它保持了全部相关的概念, 但是便于完成数字练习。考虑用图P6-2中的12位格式表示浮点数。隐含基数为2的比例因子为5位, 指数为余15格式, 其中两个端点值0和31分别表示数值0和无穷大。6位尾数使用IEEE格式规格化, 二进制小数点的左边有一个隐含的1。



图P6-2 习题6.25中的浮点数格式

- (a) 将数字+1.7、-0.012、+19和1/8表示为这种格式。  
(b) 这种格式所能表示的最小和最大值是多少?  
(c) 比较(b)部分计算的区间范围与12位有符号整数、12位有符号小数的表示范围。  
(d) 对如下的操作数A和B执行四则运算:

$$A = \begin{array}{|c|c|c|} \hline 0 & 10001 & 011011 \\ \hline \end{array}$$

$$B = \begin{array}{|c|c|c|} \hline 1 & 01111 & 101010 \\ \hline \end{array}$$

6.26 考虑用类似习题6.25格式表示的16位浮点数，它具有6位指数和9位规格化小数尾数。比例因子的基数为2，指数采用余31格式表示。

(a) 将以下格式的A和B相加：

$$A = \begin{array}{|c|c|c|} \hline 0 & 100001 & 111111110 \\ \hline \end{array}$$

$$B = \begin{array}{|c|c|c|} \hline 0 & 011111 & 001010101 \\ \hline \end{array}$$

将答案转换为规格化格式。记住二进制小数点左方隐含的1并不包含在A和B的格式中。在生成最终的9位规格化尾数时使用舍入。

(b) 使用十进制数 $w$ 、 $x$ 、 $y$ 和 $z$ 表示前面规格化浮点数格式所能表示的最大值与最小（非零）值。使用下面的格式

$$\text{最大值} = w \times 2^x$$

$$\text{最小值} = y \times 2^{-z}$$

6.27 在图6-24a浮点数表示中，指数的余 $x$ 表示为比较两个浮点数的相对大小提供了什么便利？（提示：假设有一个组合逻辑网络可以比较两个32位无符号整数的相对大小。使用这一网络以及必要的外部逻辑门设计比较浮点数所需的网络。）

6.28 在习题6.25a中，简单十进制数到二进制浮点数的转换是直接的。但是，如果十进制数是用浮点格式表示的，转换就不是直接的了，这是因为我们不能单独转换比例因子的尾数与指数，因为 $10^x = 2^y$ 一般并不能保证 $x$ 和 $y$ 为整数。假设计算机中存储着一个表，表中的二进制浮点数 $t_i$ 和 $x_i$ 的关系为 $t_i = 10^{x_i}$ 。请说明将给定的十进制浮点数转换为二进制浮点格式的一般步骤。可以使用计算机中的整数和浮点数指令。

408

6.29 将十进制数0.1表示为在6.7节开始时讨论的8位有符号二进制小数格式。如果该数字不能正好转换为8位格式，使用6.7.3节讨论的所有三种截取方法对数字作近似处理。

6.30 构建一个实例，说明当两个正数相减时获得正确的结果需要三个保护位。

6.31 习题6.2a中的四个6位答案哪些是不精确的？对每一个答案，给出与6.7.3节定义的三种截取方法相对应的三个6位值。

6.32 为图6-26中组合控制网络的输出Add/Sub（加/减）和 $S_n$ 推导逻辑等式。

6.33 如果门的扇入为4，如何组合构成图6-26中的SHIFTER（移位器）？

6.34 (a) 画出实现图6-26中的多路复用器MUX的逻辑门网络。

(b) 将图6-26中的SWAP（交换器）网络结构与(a)部分的答案相联。

6.35 如何组合实现图6-26中的前导0检测器？

6.36 图6-26中的尾数加法器/减法器对正的无符号二进制小数操作，而且必须以原码的形式给出结果。在对图6-26的讨论中我们说过对于输入和输出操作数所需的格式，反码运算是非常方便的。当两个有符号数反码相加时，必须将符号位的进位输出加到结果中才能得到正确答案。这被称为循环进位校正（end-around carry correction）。图P6-3中的两个示例使用4位

409

有符号反码对操作数和答案编码,并展示了加法操作。

当需要以原码形式生成结果时,反码运算系统是非常方便的,因为以反码形式表示的负数可以通过对符号位右方的各位求补转换为原码形式。而使用补码运算,将负数转换为原码形式时需要加入+1。如果使用超前进位加法器,可以将反码运算所需的循环进位操作结合到超前逻辑当中。请以上述讨论为指导,给出图6-26中反码加法器/减法器的完整设计。

(3) $\begin{array}{r} +(-5) \\ -2 \end{array}$	$\begin{array}{r} 0011 \\ + [0] 101100 \\ \hline 1101 \\ \hline 1101 \end{array}$	(6) $\begin{array}{r} +(-3) \\ 3 \end{array}$	$\begin{array}{r} 0110 \\ + [1] 1011000 \\ \hline 0010 \\ \hline 0011 \end{array}$
---	---	--	--

图P6-3 习题6.36中的反码加法

## 参考文献

1. Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985, August 1985.
2. J.L. Hennessy and D.A. Patterson, *Computer Architecture — A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, CA, 1996.

# 基本处理部件

### 本章目标

在本章中你将学习以下内容:

- 处理器如何执行指令
- 处理器的内部功能部件以及这些部件如何互连
- 用于产生内部控制信号的硬件
- 微程序设计方法
- 微程序结构

411

在本章和下一章我们主要讨论处理部件,该部件的主要功能是执行机器指令以及协调其他部件的活动,一般称为指令集处理器 (ISP) 或简称为处理器。下面来查看该部件的内部结构以及它是如何完成指令读取、指令译码和执行程序中的指令的。以前我们将处理器部件叫做中央处理器 (CPU), 由于许多现代计算机系统中包含有多个处理器部件, 所以今天再使用术语“中央”就有些欠妥了。

随着技术不断发展,性能要求不断提高,使得处理器的结构逐年发生变化。开发高性能处理器的一种通用方法是尽量使各种功能部件的操作能够并行处理。在高性能处理器中一般都含有流水线结构,它能够实现在前一条指令的执行结束之前开始执行另一条指令。另外一种方法称为超标量操作,它可以同时完成多条指令的读取和执行操作。流水线和超标量体系结构我们在第8章中讨论。本章将主要对各种处理器都通用的基本思想进行讨论。

一个典型的计算任务由组成程序的一系列机器指令所指定的执行步骤构成。一条指令的执行由一系列基本的操作构成,这些操作及其控制方式是本章讨论的主要内容。

### 7.1 一些基本概念

在程序的执行过程中,处理器一次取出一条指令并执行指定的操作。除非遇到转移指令或跳转指令,否则,指令从连续的存储器单元中进行读取。处理器利用程序计数器PC来跟踪将要取出的下一条指令的存储地址。指令取出以后,PC的内容将用指令序列中的下一条指令地址进行更新。但是转移指令可能会打破这种取值顺序,为PC提供一个转移到目标处的指令地址。

处理器中另一关键寄存器是指令寄存器IR。假设每条指令包含4个字节并且保存在一个存储器字中。为了执行一条指令,处理器必须完成下列三步:

1. 取出PC指向的存储器单元的内容,该单元中的内容是将要执行的指令,因而装入IR。采

用符号标记形式, 可以写成

$$IR \leftarrow [ [ PC ] ]$$

2. 假设存储器是按字节寻址的, 那么PC内容增4, 即

$$PC \leftarrow [ PC ] + 4$$

3. 执行IR中指令所规定的操作。

当一条指令超过一个字时, 必须根据需要将步骤1和2重复多次以便将指令完整地取出。这两步通常称为取指令阶段; 第3步称为执行阶段。

为了更详细地研究这些操作, 我们首先来了解处理器的内部构造。在图1-2中我们介绍过处理器的主要模块, 它们可以采用各种方法进行组织和互连。先从一个简单结构开始, 在本章的后续部分以及在第8章中将介绍一些用来提高性能的较复杂结构。图7-1给出的结构中算术逻辑部件 (ALU) 和所有寄存器是通过普通单总线互连起来的, 该总线是处理器内部总线, 不要与连接处理器和存储器及I/O设备的外部总线相混淆。

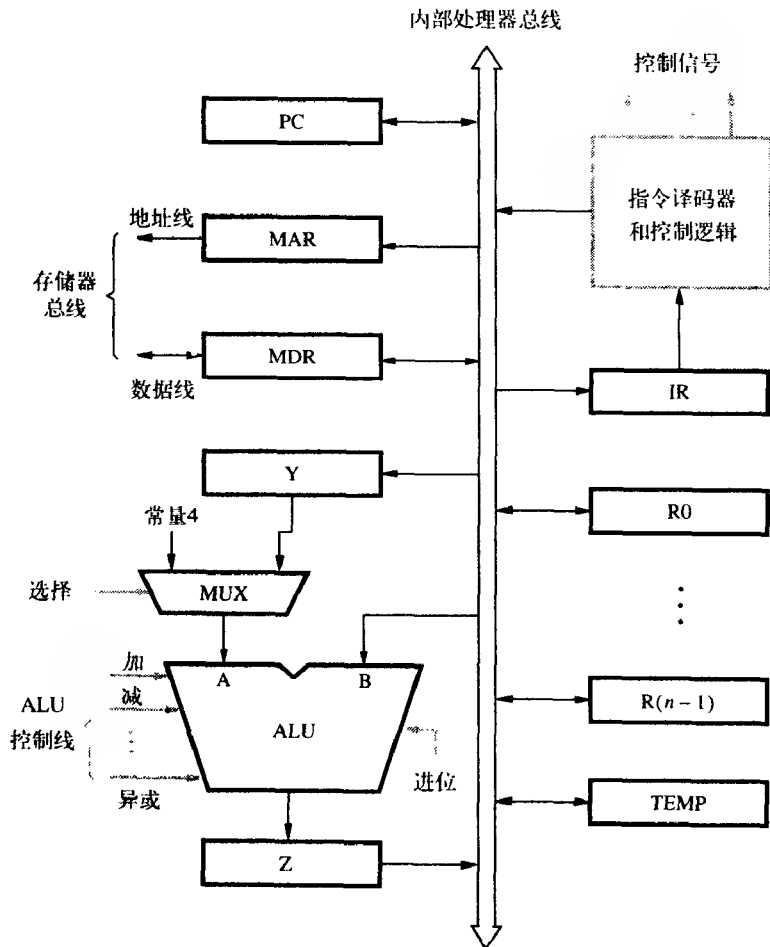


图7-1 处理器内部数据通路的单总线结构

外部存储器总线中的数据线 with 地址线通过存储器数据寄存器MDR和地址寄存器MAR分别连

接到内部处理器总线上,如图7-1所示。MDR寄存器有两个输入和两个输出,数据可以从存储器总线或内部处理器总线装入到MDR中,MDR中的数据可放在任何一条总线上。MAR的输入连接到内部总线,输出连接到外部总线。存储器总线的控制线与指令译码器和控制逻辑模块相连。控制处理器内部所有部件的操作信号以及存储器总线之间的交互信号,都是由该部件负责的。

处理器寄存器R0到R( $n-1$ )的编号和用法在不同的处理器中是各不相同的,程序员可将寄存器作为通用寄存器使用,其中某些可以指定为专用寄存器,例如变址寄存器和堆栈寄存器。图7-1中的Y、Z和TEMP三个寄存器以前没提到过。这些寄存器对程序员是透明的,即程序员不必在意它们,因为它们从未被任何指令明确地引用过。主要作用是在某些指令执行期间,由处理器作为临时寄存器使用的。这些寄存器从不用来存储由某条指令产生并在之后被其他指令使用的数据。

多路复用器MUX可以选择寄存器Y的输出或是选择作为ALU输入端A的输入常量4。这个常量4用来对程序计数器的内容增值。我们把MUX控制输入选择的两种可能值记作Select4和SelectY,分别表示选择常量4或选择寄存器Y。

在指令执行的过程中,数据从一个寄存器传送到另一寄存器时,通常要经过ALU来实现一些算术和逻辑操作。指令译码器和控制逻辑单元负责完成IR寄存器中指令所指定的操作。译码器产生所需要的控制信号来选择所涉及的寄存器并指导数据的传送。寄存器、ALU和互连总线总体被称为数据通路。

除了少量情况以外,一条指令可以通过按某种指定的顺序执行以下一个或多个操作来完成:

414

- 从一个处理器寄存器向另一寄存器或ALU传送一个字长的数据。
- 执行算术或逻辑运算并将结果保存到处理器寄存器中。
- 取出指定存储器单元的内容将其装入到处理器寄存器中。
- 将处理器寄存器中一个字长的数据保存到指定的存储单元中。

现在我们采用图7-1的简单处理器模型,具体考虑这些操作中的每一种操作是如何实现的。

### 7.1.1 寄存器传送

指令执行中包括一系列操作步骤,这期间数据在寄存器之间不断传送。每个寄存器都使用两种控制信号,其中一种信号用来将寄存器的内容放到总线上,另一种信号用来将总线上的数据装入寄存器中。在图7-2中我们给出了表示的符号。寄存器 $R_i$ 的输入和输出分别经由信号 $R_{i_{in}}$ 和 $R_{i_{out}}$ 所控制的开关连接到总线上。当 $R_{i_{in}}$ 置1时,总线上的数据装入到 $R_i$ 中。同样,当 $R_{i_{out}}$ 置1时,寄存器 $R_i$ 的内容被放到总线上。当 $R_{i_{out}}$ 等于0时,总线用来传送其他寄存器的数据。

假设我们希望将寄存器R1的内容传送到R4中。可以采用如下方法来完成:

- 通过将 $R_{1_{out}}$ 置成1使寄存器R1可以输出,这样可以将R1的内容放到处理器总线上。
- 通过将 $R_{4_{in}}$ 置为1使寄存器R4可以输入,这样可以将处理器总线上的数据装入到寄存器R4中。

处理器内部的所有操作和数据传送都在由处理器时钟给定的时间周期内完成。在时钟周期的起始处判断传送的具体控制信号,本例中是将 $R_{1_{out}}$ 和 $R_{4_{in}}$ 置为1。寄存器由边沿触发器构成。因此,在下一个时钟的活动边沿,构成R4的触发器装入输入端的数据。同时,控制信号 $R_{1_{out}}$ 和 $R_{4_{in}}$ 返回0。在本章的其余部分我们将使用这种定时数据传送的简单模型。不过,应该指出的是使用其他方法也是可行的。例如,数据传送可以使用时钟的上升沿和下降沿。而且,当不用边

沿触发器时可能需要两个或更多的时钟信号来确保正确的数据传送。这种方法称为多相时钟。

对于寄存器 $R_i$ 中的一位实现如图7-3所示。一个两输入端的多路复用器用来选择边沿D触发器的输入端数据。当控制输入 $R_{i_{in}}$ 等于1时,多路复用器选择总线上的数据,该数据在时钟的上升沿被装入到触发器中。当 $R_{i_{in}}$ 等于0时,多路复用器反馈当前保存在触发器中的值。

触发器Q输出端经由三态门与总线相连。当 $R_{i_{out}}$ 等于0时,三态门的输出端呈高阻态(电路断开),这与开关的开路状态一致。当 $R_{i_{out}}=1$ 时,三态门根据Q值来驱动总线为0或为1。

### 7.1.2 执行算术或逻辑操作

ALU是不含内部存储器的组合电路。它完成对A和B输入端的两个操作数的算术和逻辑运算。在图7-1和图7-2中,一个操作数是多路复用器MUX的输出,另一个操作数直接从总线上获得。ALU产生的结果临时存储在寄存器Z中。因此,将寄存器R1的内容与R2的内容相加后结果存在寄存器R3中的操作序列是:

1.  $R1_{out}, Y_{in}$
2.  $R2_{out}, \text{SelectY}, \text{Add}, Z_{in}$
3.  $Z_{out}, R3_{in}$

某一步中命名的信号在其相应那一步的时钟周期内是处于激活状态的,而所有其他信号是不被激活的。因此,第1步,允许寄存器R1输出以及寄存器Y输入,使R1的内容由总线传送到Y。在第2步,多路复用器的选择信号设置为SelectY,使多路复用器控制寄存器Y的内容传送到ALU的A输入端上。同时,寄存器R2的内容传送到总线,然后再输入到B输入端上。ALU实现的功能取决于该部件的控制线上使用的信号。在这种情况下,Add线被置成1,将ALU的A、B两个输入端的数相加,并输出相加的和。由于它的输入控制信号是激活状态,所以将这个相加的和装入到寄存器Z中。第3步,将寄存器Z的内容传送到目标寄存器R3中。由于在每个时钟周期中间,只有一个寄存器输出连接到总线上,所以最后一步的传送不能在第2步中完成。

在这里的介绍性讨论中,我们假设要执行的每一功能都有一个专用信号。例如,假设有独立的控制信号来表示单个ALU操作,如Add、Subtract、XOR等。在实际的应用中,采用一定位数的编码来表示。例如,如果ALU可实现八种不同操作,那么需要使用三种控制信号就足以表示所需要的操作。在7.5.1节我们

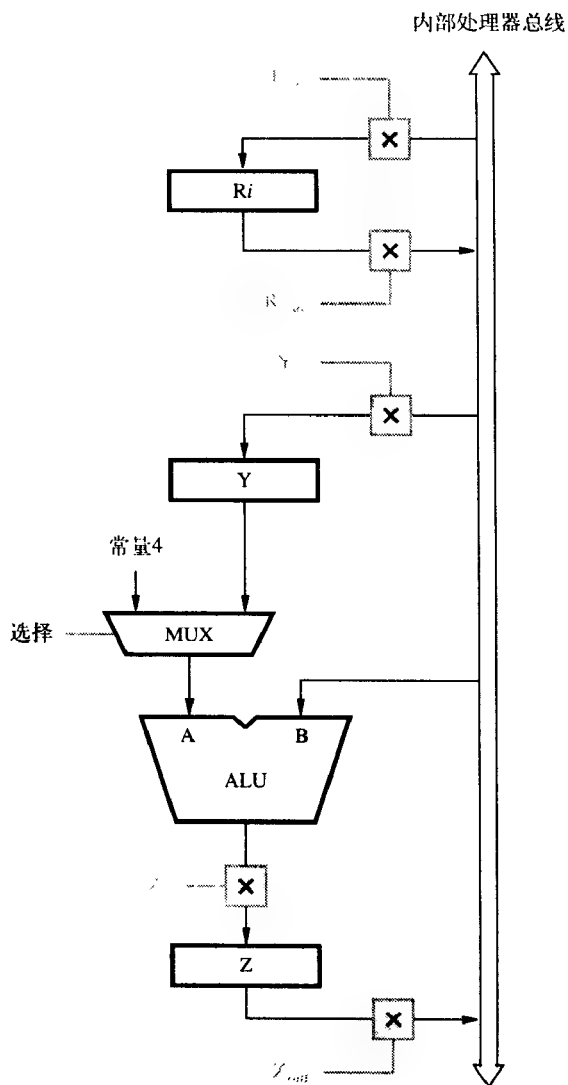


图7-2 图7-1中寄存器的输入和输出控制

将讨论控制信号编码的限制以及利弊。

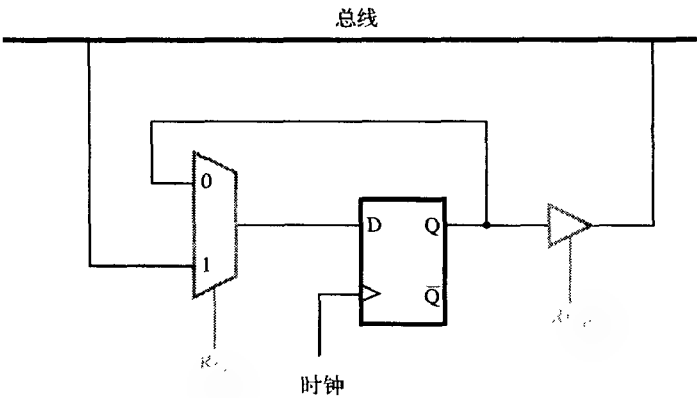


图7-3 寄存器中一位的输入和输出控制

417

7.1.3 从存储器中取出一个字

为了从存储器中取出一个字长的信息，处理器必须指明存放该信息的存储器地址并请求读操作。无论要取的信息表示程序指令还是由指令指明的操作数要求这样做。处理器将所请求的地址传送到MAR，MAR输出连接到存储器总线的地址线上。同时，处理器使用存储器总线的控制线指明需要的读操作。当从存储器中接收到所需要的数据时，便将该数据保存到寄存器MDR中，还可以将MDR寄存器中保存的这些数据传送到处理器的其他寄存器中。

寄存器MDR的连接如图7-4所示。MDR有四个控制信号： $MDR_{in}$ 和 $MDR_{out}$ 控制与内部总线的连接， $MDR_{inE}$ 和 $MDR_{outE}$ 控制与外部总线的连接。对图7-3中的电路稍微修改便可以提供额外的连接。可用三输入多路复用器将存储器总线的数据线连接到第三个输入端上。当 $MDR_{inE} = 1$ 时选择该输入端，由 $MDR_{outE}$ 控制的一个三态门用来连接触发器的输出端和存储器总线。

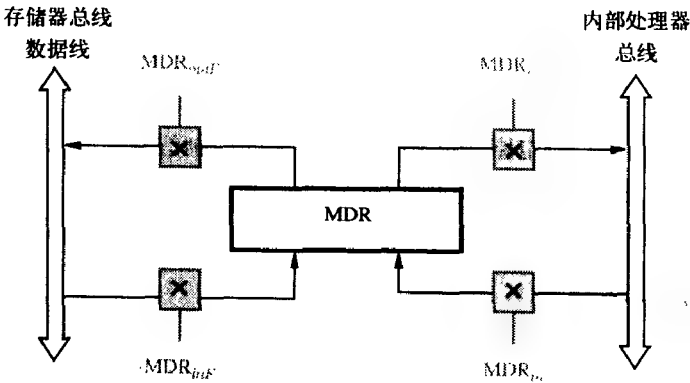


图7-4 寄存器MDR的连接及控制信号

418

在进行存储器读和写操作期间，内部处理器操作的时序必须与存储器总线上寻址设备的响应保持一致。处理器在一个时钟周期内完成一个内部数据传送，而寻址设备的操作速度随设备的不同而有所差异。我们在第5章中看到现代处理器在处理器芯片中包括一个片上的高速缓存（cache）。通常，高速缓存存在一个时钟周期中响应一次存储器的读请求。然而，当发生高速缓存未命中情况时，读请求传送到主存储器上，这将会导致有多个时钟周期的延迟。读/写



请求还可以使用预定义的存储器映射I/O设备中的寄存器。此类I/O寄存器没有高速缓存,所以访问这些寄存器总是需要花费大量时钟周期的。

为了适应响应时间的不同,处理器将一直等待,直到它接收到所请求的读操作已经完成的提示为止。为了达到这个目的,我们将假设使用一个叫做存储器功能完成(MFC)的控制信号。寻址设备将这个信号设置为1来表示指定单元的内容已经读出并且已经放到了存储器总线的数据线上。(结合第4章对总线的讨论,我们已经遇到过此类信号的多个例子,例如图4-25的从动就绪以及图4-41中的TRDY#。)

看一个读操作的例子,考虑指令 Move (R1), R2。执行这条指令所需要的动作是:

1.  $MAR \leftarrow [R1]$
2. 在存储器总线上开始一个读操作
3. 等待存储器的MFC响应
4. 从存储器总线装入MDR
5.  $R2 \leftarrow [MDR]$

这些动作可以作为独立的步骤来执行,但有些也可以合并成一步来完成。每个动作可以在一个时钟周期中完成,只有动作3需要一个或更多时钟周期,这要取决于寻址设备的速度。

为了简单起见,我们假设MAR的输出一直处于使能状态,因此MAR的内容总是可以在存储器总线的地址线上获得。这种情况是处理器作为总线控制器的情况。当一个新地址装入到MAR时,就在下一时钟周期的开始时出现在存储器总线上,如图7-5所示。在地址装入MAR的同时激活了读控制的信号,该信号使总线接口电路向总线上发出读请求MR。使用这种安排,我们可以将上面的动作1和2合并成一个控制步骤。当等待存储器响应时,通过激活控制信号MDR<sub>inE</sub>将动作3和4也组合在一起。因而,从存储器收到的数据在MFC信号收到的那个时钟周期的末尾装入MDR中。在下一时钟周期,激活MDR<sub>out</sub>将数据传送到寄存器R2中。这意味着存储器读操作需要三步,使用被激活的信号来描述如下:

419

1. R1<sub>out</sub>, MAR<sub>in</sub>, Read
2. MDR<sub>inE</sub>, WMFC
3. MDR<sub>out</sub>, R2<sub>in</sub>

其中WMFC是处理器的控制电路等待MFC信号到来的控制信号。

图7-5表明,就在读命令的同一个周期MR内MDR<sub>inE</sub>被置成1。因此,在后续的讨论中,我们并不明确指定MDR<sub>inE</sub>的值,并认为它总是等于MR的。

#### 7.1.4 向存储器中存储一个字

向存储器中存储入一个字遵从类似的过程:将目标地址装到MAR中,接着将要写入的数据装入到MDR中并发出写请求。因此,执行指令Move R2, (R1)时的顺序如下:

1. R1<sub>out</sub>, MAR<sub>in</sub>
2. R2<sub>out</sub>, MDR<sub>in</sub>, Write
3. MDR<sub>outE</sub>, WMFC

420

和读操作一样,写控制信号使存储器总线接口硬件向存储器总线发送写请求。处理器停留在第3步,直到存储器操作完成并收到MFC响应为止。

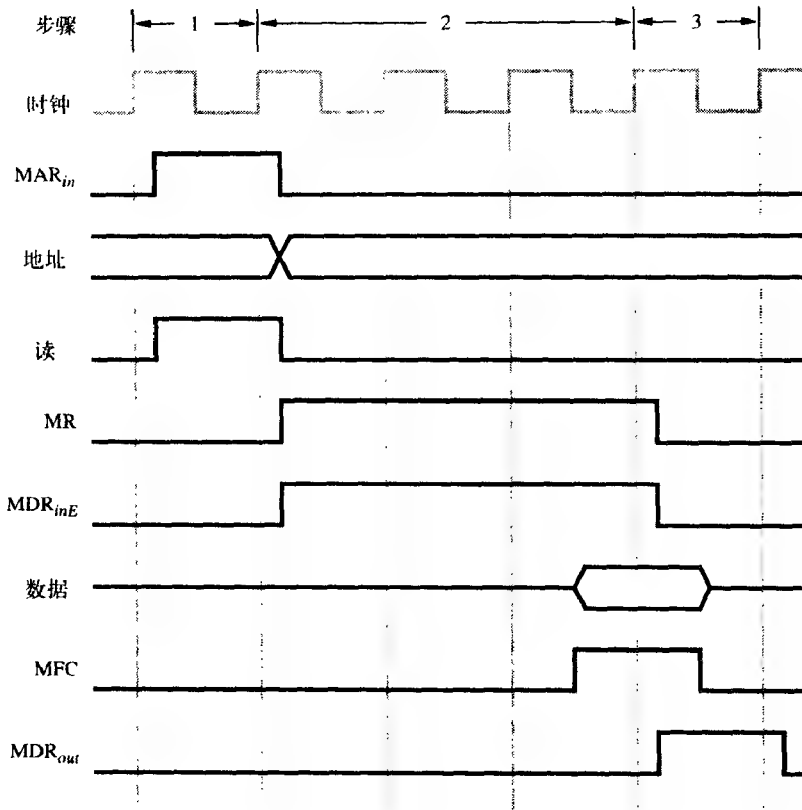


图7-5 存储器读操作的时序

## 7.2 一条完整指令的执行

现在让我们把执行一条指令所要求的一系列基本操作结合在一起。下面考虑指令

Add (R3), R1

该指令将R3所指向的存储器单元的内容与寄存器R1的内容相加。执行该指令需要如下动作:

1. 取指令
2. 取第一个操作数 (R3所指向的存储器单元中的内容)
3. 执行加法操作
4. 结果存入R1

图7-6给出了在图7-1的单总线体系结构中, 实现这些操作所需要的一系列控制步骤。指令执行过程如下。在第1步中, 通过将PC内容装到MAR中以及向存储器发送读请求, 初始化取指令操作。选择信号设置为Select4, 使多路复用器MUX选择常量4。该值与B输入端的操作数 (即PC的内容) 相加, 结果保存到寄存器Z中。在第2步中, 当等待的存储器响应时, 将寄存器Z中更新后的地址传送回PC中。第3步, 将从存储器中取出的字装入到IR中。

第1步到第3步构成取指令阶段, 这对所有指令都是相同的。指令译码电路在第4步的开始便解释IR的内容, 这样可以使控制电路激活第4步到第7步中的控制信号, 第4步到第7步构成指令的执行阶段。在第4步中, 寄存器R3的内容传送到MAR, 并开始存储器读操作。接着在第5步, R1的内容传送到寄存器Y中, 为加法操作做准备。当读操作完成时, 在寄存器MDR中获得存储器操

作数，在第6步执行加法操作。MDR的内容送到总线上，这样也就送到了ALU的B输入端。通过选择SelectY，可选择寄存器Y作为ALU的第2个输入端。相加的和保存到寄存器Z中，之后在第7步送到R1中。End信号将用返回到第1步的方式开始一条新指令的读取周期。

在这个讨论中阐述了图7-6除了第2步中 $Y_{in}$ 以外的所有控制信号。在执行加法指令中，不需要将被更新的PC内容复制到寄存器Y中。  
但是，在转移指令中，需要使用PC的更新值来计算转移的目标地址。为了加速转移指令的执行，在第2步中将该值复制到寄存器Y中。由于第2步是取指令阶段的一部分，所以对于所有指令会执行相同的动作。由于寄存器Y并不用于其他目的，所以并不会带来任何影响。

步骤	动作
1	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
2	$Z_{out}, PC_{in}, Y_{in}, WMFC$
3	$MDR_{out}, IR_{in}$
4	$R3_{out}, MAR_{in}, Read$
5	$R1_{out}, Y_{in}, WMFC$
6	$MDR_{out}, SelectY, Add, Z_{in}$
7	$Z_{out}, R1_{in}, End$

图7-6 执行Add (R3), R1 指令的控制序列

转移指令

转移指令用转移目标地址代替PC中的内容，该地址一般通过将PC的值与转移指令给出的偏移量X相加而获得被更新的值。图7-7给出了实现一个无条件转移指令的控制序列。通常从取指令阶段开始处理，到第3步指令装入IR时，取指令阶段结束。指令译码电路从IR中分离出偏移量，如果需要，指令译码电路要进行符号扩展。由于PC的更新值已存入寄存器Y中，所以在第4步将偏移量X送到总线上，并且进行加法操作。在第5步将操作的结果转移目标地址装入PC中。

步骤	动作
1	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
2	$Z_{out}, PC_{in}, Y_{in}, WMFC$
3	$MDR_{out}, IR_{in}$
4	$Offset-field-of-IR_{out}, Add, Z_{in}$
5	$Z_{out}, PC_{in}, End$

图7-7 一条无条件转移指令的控制序列

转移指令用到的偏移量X，通常是转移目标地址和直接跟在转移指令后的指令的地址之差。例如，如果转移指令在地址为2000的单元中，转移目标地址是2050，那么X值一定是46。这个推断可以立即从图7-7的控制序列中看出来。在取指令阶段，在知道所执行指令的类型之前，PC被增值。因此，当在第4步计算转移地址时，使用的PC值是更新后的值，它指向的是存储器中转移指令之后的那条指令。

现在考虑条件转移指令。在这种情况下，我们需要在PC装入新值之前检查条件码的状态。例如，对于一个负数转移 (Branch<0) 指令，图7-7的第4步用下式来替换：

$Offset-field-of-IR_{out}, Add, Z_{in}, If\ N = 0\ then\ End$

因此，如果 $N = 0$ ，处理器在第4步后返回到第1步。如果 $N=1$ ，执行第5步将新值装入到PC中，从而实现转移操作。

7.3 多总线结构

我们使用图7-1中简单的单总线结构来说明基本思想。由于在一个时钟周期内，只能有一个数据项在总线上传输，所以图7-6和图7-7中的最终控制序列很长。为了减少所需要的步骤，大多

数商业处理器提供允许多个传输并行执行的多个内部通路。

图7-8描述了用于连接处理器寄存器和ALU的三总线结构。所有通用寄存器组合成一个独立的块，称为寄存器文件。在VLSI工艺中，实现大量寄存器的最有效方法是存储单元阵列的形式，类似于第5章所描述的用于实现随机访问存储器（RAM）的那些方法。上面所说的图7-8中的寄存器文件具有3个端口。其中两个是输出端口，允许同时访问两个不同寄存器的内容，并把寄存器内容放在总线A和总线B上；第三个端口允许总线C上的数据在同一时钟周期内装入到第三个寄存器中。

总线A和总线B用于将源操作数传送到ALU的A输入端和B输入端上，在ALU中可实现算术或逻辑运算，通过总线C将结果传送到目的地。如果需要，ALU可以将两个输入操作数中的一个不加修改地传送给总线C。我们为此类操作（ $R=A$ 或 $R=B$ ）调用ALU控制信号。三总线结构可以省略图7-1中的寄存器Y和Z。

图7-8的第二个特征是引入了递增部件，它用于对PC每次加4。使用递增部件省略了如图7-6和图7-7中需要使用主ALU完成PC加4的方法。在ALU输入端的多路复用器的常量4的源还是有用的，它可用于递增其他的地址，例如在LoadMultiple和StoreMultiple指令中的存储器地址。

考虑三操作数指令

Add R4, R5, R6

在图7-9中给出了执行该指令的控制序列。第1步，利用 $R=B$ 的控制信号将PC的内容经过ALU装入到MAR中以启动存储器的读操作。同时PC递增4。注意装入到MAR的值是PC中原来的值。加载到PC中的增量值是在时钟周期末完成的，所以不会影响MAR的内容。第2步，处理器等待MFC并把收到的数据装入MDR中，接着在第3步把这些数据送到IR。最后，指令的执行阶段只需控制步中的一个第4步便可以完成。

通过为数据传输提供更多的通路，大大减少了指令执行时所需要的时钟周期数量。

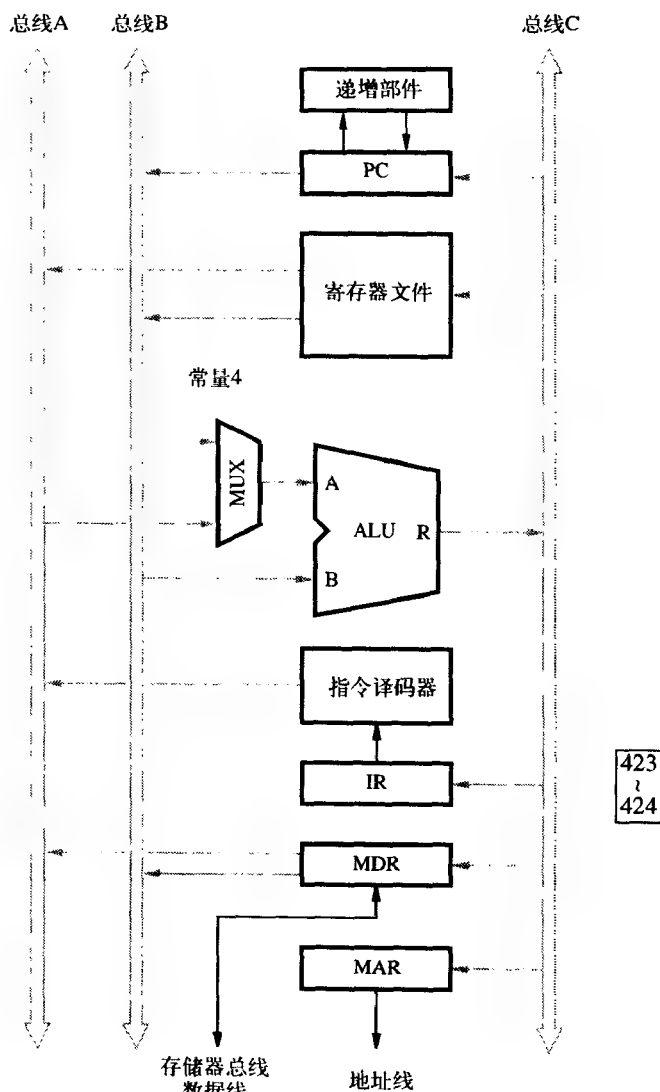


图7-8 数据通路的三总线结构

步骤	动作
1	PC <sub>out</sub> , R=B, MAR <sub>in</sub> , Read, IncPC
2	WMFC
3	MDR <sub>outB</sub> , R=B, IR <sub>in</sub>
4	R4 <sub>outA</sub> , R5 <sub>outB</sub> , SelectA, Add, R6 <sub>in</sub> , End

图7-9 对于图7-8的三总线结构，指令Add R4, R5, R6的控制序列

## 7.4 硬件控制

处理器为了执行指令，必须具备一些可以按正确顺序产生所需控制信号的方法。计算机设计者使用大量技术来解决这类问题。基本上有两种方法：硬件控制和微程序控制。下面我们对每一种技术进行详细的讨论，本节首先讨论硬件控制。

考虑图7-6给出的控制信号序列。该序列中的每一步在一个时钟周期内完成。可以使用一个计数器来跟踪控制步，如图7-10所示。计数器中的每一个状态或每个计数对应着一个控制步。所需要的控制信号由下列信息决定：

- 控制步计数器的内容
- 指令寄存器的内容
- 条件码标志的内容
- 外部输入信号，比如MFC和中断请求

为了进一步了解控制器的组成，我们首先看一下相关的硬件简化图。图7-10中的译码器/编码器块是根据所有输入状态产生所需要的控制输出的组合电路。通过将译码和编码功能分离，我们得到了更具体的框图7-11。步译码器按控制顺序为每一步或每个时间片提供独立的信号线。类似地，指令译码器的输出是由每条机器指令对应一条单独输出线构成的。对于装入IR中的任何一条指令，输出线 $INS_i$ 到 $INS_m$ 中只能有一条线设置成1，其他输出线全部设置成0（译码器的设计细节参看附录A）。图7-11中编码器块将输入信号组合并产生单独的 $Y_{in}$ 、 $PC_{out}$ 、Add、End等控制信号。对于图7-1中的处理器结构，我们在图7-12中给出一个如何采用编码器来产生 $Z_{in}$ 控制信号的实例。该电路实现逻辑函数

$$Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \dots \quad (7-1)$$

对于所有指令，该信号在时间段 $T_1$ 产生，对于ADD指令在 $T_6$ 产生，对于无条件转移指令在 $T_4$ 产生，等等。 $Z_{in}$ 的逻辑函数从图7-6和图7-7中的控制序列中获得。我们再看一个例子，图7-13给出由逻辑函数

$$End = T_7 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \bar{N}) \cdot BRN + \dots \quad (7-2)$$

产生End控制信号的电路。End信号通过将控制步计数器的值重新设置成它的初始值来启动新的取指令周期。在图7-11中包含另一称作RUN的控制信号，当RUN等于1时，使计数器在每个时钟周期的结尾增1。当RUN等于0时，计数器停止计数。每当要发送WMFC信号时，处理器都要等待来自存储器的响应。

图7-10或图7-11所示的控制硬件可被看作是一个状态机，它根据指令寄存器中的内容、条件码和外部输入，在每个时钟周期内，从一种状态变换到另一种状态。状态机的输出是控制信号。该状态机所执行的操作序列是由逻辑单元的线路所决定的，因而称为“硬件”。采用这种方法的控制器能够以很高的速度进行操作。但是，这种方法的灵活性较差，可实现指令集的复杂度也是有限的。

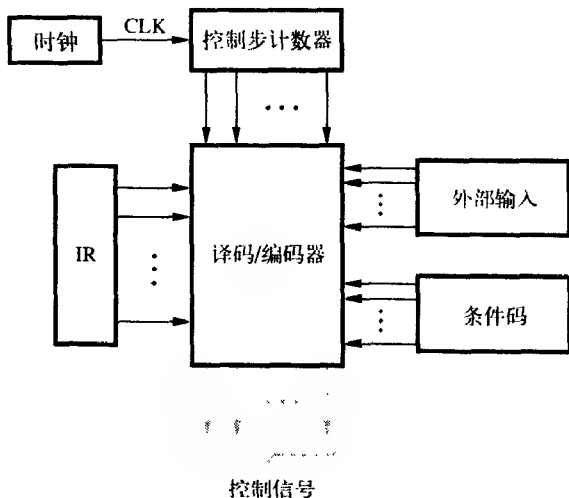


图7-10 控制器的组成

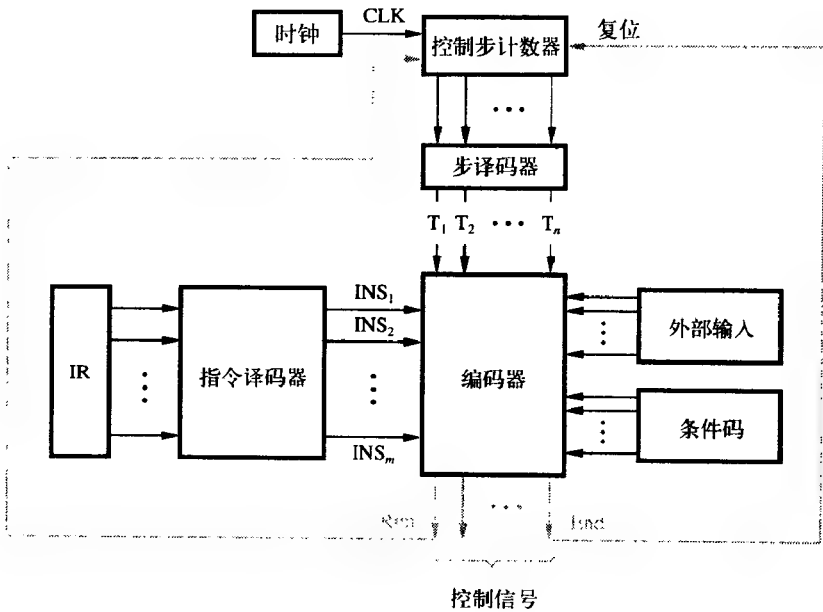


图7-11 分离的译码和编码功能

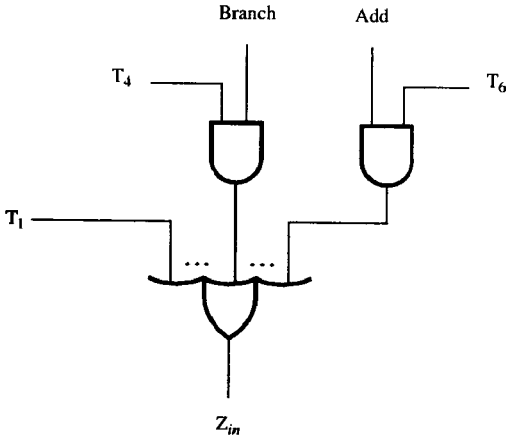


图7-12 图7-1处理器Z<sub>in</sub>控制信号的产生

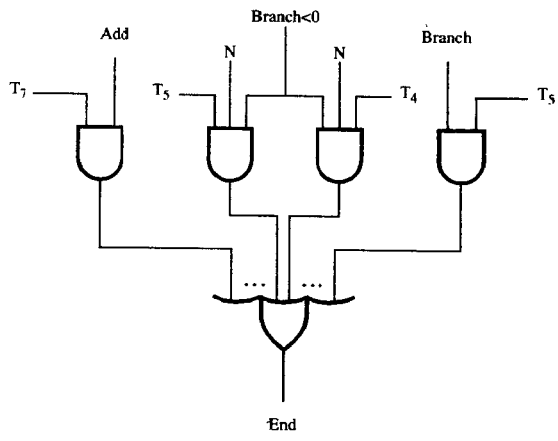


图7-13 End控制信号的产生

427

一个完整的处理器

可以用图7-14所示的结构来设计一个完整的处理器。此结构包含有一个指令部件，该部件可以从指令高速缓存中读取指令，而当所需的指令不在缓存中时可以立即从主存中读取指令。在该处理器中具有单独的部件分别处理整数和浮点数。其中，每个部件都可以组织成图7-8所示的结构。在这些部件与主存之间插入了一个数据高速缓存。目前有许多处理器采用分离的指令和数据高速缓存。不过也有使用一个缓存来存储指令和数据的处理器。处理器是与系统总线相连的，因而计算机中的其他部件通过总线接口与处理器相连。

428

尽管在图7-14中只画出了一个整数部件和一个浮点部件，为了提高机器的并行处理能力，通常在处理器中会包含有多个不同类型的部件。使用多个部件来提高指令执行速度的方法，我们将在第8章中讨论。

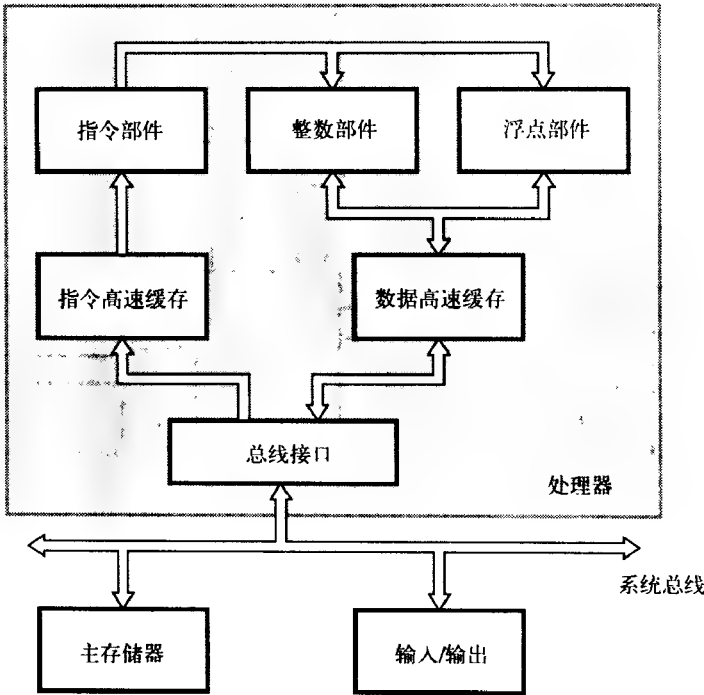


图7-14 一个完整的处理器框图

7.5 微程序控制

在7.4节中，我们看到处理器内部所需要的控制信号是如何使用控制步计数器和译码器/编码器电路产生的。现在讨论另一种称为微程序控制的技术，这里控制信号由类似于机器语言的程序来产生。

429

微指令	:	PC <sub>in</sub>	PC <sub>out</sub>	MAR <sub>in</sub>	Read	MDR <sub>out</sub>	IR <sub>in</sub>	Y <sub>in</sub>	Select	Add	Z <sub>in</sub>	Z <sub>out</sub>	R1 <sub>out</sub>	R1 <sub>in</sub>	R3 <sub>out</sub>	WMFC	End	:
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	

图7-15 用于图7-6的微指令实例

首先介绍一些常用术语。控制字（CW）是一个字，它的每一位代表图7-11中的不同控制信号。指令控制序列中的每一步控制在CW中采用1或0的方式来定义。对于图7-6中的7个步骤所对应的CW情况如图7-15所示。我们假设SelectY由Select=0表示，Select4由Select=1表示。由机器指令的控制序列所对应的一系列CW构成该指令的微程序，在该微程序中的单个控制字称为微指令。

一台计算机指令集中的所有指令微程序存储在一个特定的存储器中, 该存储器称为控制存储器。控制部件通过从控制存储器中连续读取相应微程序的CW, 可以为任何一个指令产生控制信号。图7-16中给出了一个控制部件组织结构的建议形式。为了从控制存储器中连续地读取控制字, 这里引用了一个微程序计数器 ( $\mu PC$ )。每当有新的指令装入到IR中时, 被称为“启动地址产生器”的模块的输出便装入到 $\mu PC$ 中。之后 $\mu PC$ 便随时钟自动增加, 这样就可以从控制存储器中连续地读出微指令。因此, 控制信号就可以按照正确的顺序传送到处理器的各个部分中。

430

图7-16中使用的简单结构无法实现控制部件的一个重要功能, 即当需要控制器需要检查条件码的状态, 或对外部输入进行检查以便从两个动作过程中选择一个执行的功能无法形成。在硬件控制情况下, 这种功能可以通过在编码器电路中增加一个适当的形如7-2等式所示的逻辑函数来完成。在微程序控制情形下, 一种可选的方法是使用条件转移微指令。这些微指令除了指定转移地址外, 还将指明外部输入、条件码或指令寄存器中的位, 这些内容是产生转移的条件。

现在, 指令Branch<0可以用图7-17所示那种微程序来实现了。将该指令读取到IR之后, 转移微指令将控制权转给相应的微程序, 我们假定在控制存储器的单元25处开始。那么, 这个地址就是图7-16中的启动地址产生器模块的输出。在单元25处的微指令将测试条件码的第N位, 如果该位等于0, 就转移到单元0处取新的机器指令; 否则, 执行在单元26处的微指令, 将转移目标地址放入寄存器Z中, 如同图7-7中的第4步。在单元27处的微指令将该地址送到PC中。

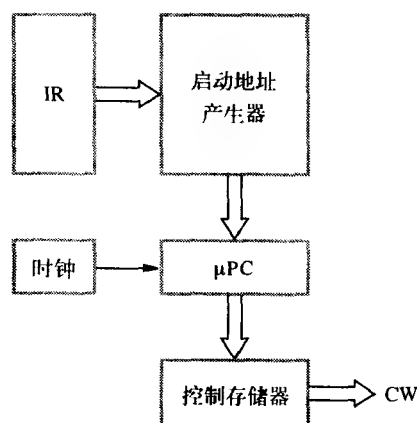


图7-16 微程序控制部件的基本构成

431

地址	微指令
0	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
1	$Z_{out}, PC_{in}, Y_{in}, WMFC$
2	$MDR_{out}, IR_{in}$
3	Branch to starting address of appropriate microroutine
.....	.....
25	If $N=0$ , then branch to microinstruction 0
26	Offset-field-of- $IR_{out}, SelectY, Add, Z_{in}$
27	$Z_{out}, PC_{in}, End$

图7-17 指令Branch&lt;0的微程序

为了支持微程序的转移, 将控制器的结构修改成如图7-18所示的情形。图7-16中的起始地址生成器变成了启动和转移地址产生器。该模块在微指令的驱动下, 将新地址读入到 $\mu PC$ 中。为了能够实现条件转移, 将外部输入、条件码以及指令寄存器的内容作为该模块的输入。在这种控制器中, 每当从微程序存储器中取出一条新的微指令时 $\mu PC$ 就会递增。不过以下的几种情况例外:

1. 当新指令取入IR时,  $\mu PC$ 中装入了这条指令的微程序首地址。



2. 当遇到转移微指令且转移条件满足时,  $\mu PC$ 被装入了转移地址。

3. 当遇到End微指令时,  $\mu PC$ 被装入了取指令周期的微程序中的第一个CW的地址(在图7-17中该地址是0)。

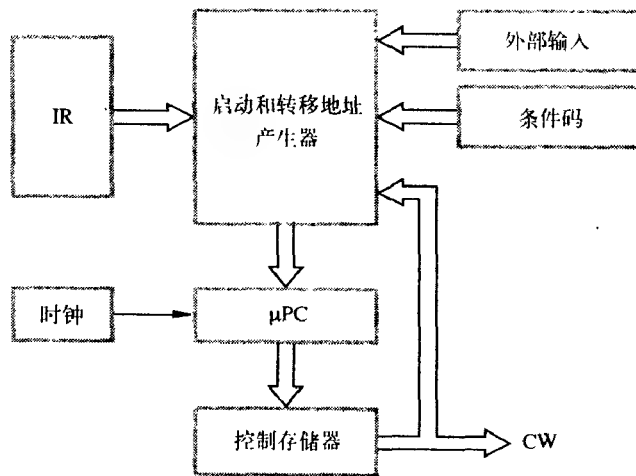


图7-18 允许在微程序中条件转移的控制部件的结构

### 7.5.1 微指令

**432** 在前面我们对串行微指令技术进行了描述, 现在进一步了解单条微指令的格式。构造微指令的直接方法是为每个控制信号分配一位的位置, 如图7-15所示。然而, 这种方法有一个严重的缺陷——由于所需信号量通常很大, 所以给每个控制信号分配一位会使微指令变得很长。而且, 在任何指定的微指令中, 只有很少几位被设置成1(用于主动控制), 这意味着可用的位空间是非常有限的。再考虑图7-1的简单处理器, 假设它只包含四个通用寄存器R0、R1、R2和R3。在该处理器中有些连接是始终被允许的, 例如IR到译码电路的输出以及ALU的两个输入都是此种情形。其余到各个寄存器的连接总共需要20个控制信号。其他所需的控制信号在图中没有给出, 其中包括读、写、选择、WMFC和结束信号。最后必须说明ALU所能实现的功能。假设提供16种功能, 包括加、减、与和异或。这些功能取决于所使用的特定ALU, 而不必与机器指令操作码一一对应。所以, 这里总共需要42个控制信号。

如果采用前面描述的简单编码技术, 那么每条微指令需要42位。幸运的是, 微指令的长度可以很容易地进行压缩。其中, 大多数信号并不是同步的, 并且还有许多信号之间是互相排斥的。例如, 一次只能激活ALU的一种功能。传输的数据源必须是惟一的, 因为同时不可以将两个不同寄存器的内容传送到总线上。另外, 存储器的读和写信号不能同时被激活。这就表示可以将信号分组, 以便所有的互斥信号归于同一组中。因此, 在任何一条微指令中, 每组中至多只能指定一条微操作。于是, 可以采用二进制编码技术来表示同一组中的信号。例如, 对于ALU中的16种功能, 用四位就足够表示了。寄存器输出控制信号 $PC_{out}$ 、 $MDR_{out}$ 、 $Z_{out}$ 、 $Offset_{out}$ 、 $R0_{out}$ 、 $R1_{out}$ 、 $R2_{out}$ 、 $R3_{out}$ 和 $TEMP_{out}$ 可以归为同一组。其中的任何一种信号可以用惟一的4位代码来选定。

对于其余信号可以作进一步的自然分组。图7-19是微指令中部分格式的例子, 其中每个分组所占用的字段要足够容下所需要的代码。在大多数字段中, 必须要包括一个不需要这一动作

的非激活代码。例如，F1中的全0方式表示在F1字段中可表示的所有寄存器中，没有一个寄存器需要将其内容放到总线上。并不是在所有字段中都需有非激活代码，例如，F4包含4位，表示ALU中实现的16种操作。因为不包括空代码，所以ALU在每一微指令执行期间都是被激活的。然而，它的动作要通过寄存器Z受到机器其余部分的监控。只有激活Z<sub>in</sub>信号时，才将ALU的输出装入到寄存器Z中。

将控制信号按字段分组需要一些硬件，因为译码器电路必须将每个字段中的位方式译码成单个的控制信号。这个额外硬件的开销要远远高于为了减少控制存储对每条微指令的位数进行压缩的开销。在图7-19中，42个信号的模式只需要20位就可以存储。

到目前为止，我们只是考虑了互斥控制信号的分组和编码。可以将这种思想进行扩展，列举出所有可能的微指令中所需信号的模式。接着可以对动作控制信号中的每一个有意义的组合分配一个不同的代码，用来表示那条微指令。这种完全编码方式可能会进一步压缩微指令的长度，不过同时也会增加所需译码器电路的复杂程度。

433

微指令				
F1	F2	F3	F4	F5
F1 (4位)	F2 (3位)	F3 (3位)	F4 (4位)	F5 (2位)
0000: No transfer	000: No transfer	000: No transfer	0000: Add	00: No action
0001: PC <sub>out</sub>	001: PC <sub>in</sub>	001: MAR <sub>in</sub>	0001: Sub	01: Read
0010: MDR <sub>out</sub>	010: IR <sub>in</sub>	010: MDR <sub>in</sub>	⋮	10: Write
0011: Z <sub>out</sub>	011: Z <sub>in</sub>	011: TEMP <sub>in</sub>	⋮	
0100: R0 <sub>out</sub>	100: R0 <sub>in</sub>	100: Y <sub>in</sub>	1111: XOR	
0101: R1 <sub>out</sub>	101: R1 <sub>in</sub>		⏟	
0110: R2 <sub>out</sub>	110: R2 <sub>in</sub>		16种ALU	
0111: R3 <sub>out</sub>	111: R3 <sub>in</sub>		功能	
1010: TEMP <sub>out</sub>				
1011: Offset <sub>out</sub>				
F6	F7	F8	...	
F6 (1位)	F7 (1位)	F8 (1位)		
0: SelectY	0: No action	0: Continue		
1: Select4	1: WMFC	1: End		

图7-19 字段编码微指令部分格式的实例

采用压缩代码来表示每条微指令中的少量控制功能的高级编码方法，称为纵向结构。相反的是图7-15的最少化编码方案，它将许多资源用一条微指令控制，被称为是横向结构。当希望更快的操作速度以及机器结构允许并行使用资源时，横向方法是非常有效的。由于纵向方法在实现所需的控制功能时要使用较多的微指令，所以它的操作速度相对来说要慢一些。尽管每条微指令需要很少的位数，但并不表示控制存储器中的全部位数会更少。其中的关键因素是需要较少的硬件来控制微指令执行。

434

在微程序控制中, 横向结构和纵向结构代表了两种极端组织方式。还可以采用许多介于这两者之间的方案, 其中编码度是一个设计参数。图7-19中的布局是横向结构, 因为它只把互斥微操作分在同一字段中。结果它丝毫没有限制处理器并行实现各种微操作的能力。

虽然我们只考虑了所有可能控制信号的一个子集, 不过这个子集可以代表实际的要求。这里省略了对理解操作原理没有帮助的一些细节。

### 7.3.2 微程序的顺序

在图7-15中的简单微程序实例中, 除了在取指令阶段最后的转移以外, 只需要简单顺序执行微指令。如果每条机器指令都是由这种微程序实现的, 那么图7-18中给出的微控制结构就足够了。在该结构中 $\mu$ PC控制顺序, 通过将机器指令译码成装入 $\mu$ PC的首地址来进入微程序。微程序内的一些转移能力可以通过指定转移地址的专用转移微指令引入, 类似于机器级指令进行转移的方法。

有了这种方法, 编写微程序就相当简单了, 因为可以使用标准的软件技术。然而, 这种方法有两个主要缺点。由于每条机器指令都有独立的微程序, 所以微指令的数量会很多, 并且会占用大量的控制存储器。如果大部分机器指令都包含多种寻址方式, 就会存在许多指令与寻址方式的组合。每种组合都会有一个独立的微程序, 还会产生相同部分的副本。我们希望将微程序组织成尽可能多共享相同部分的结构。这要求有许多转移微指令在各个部分之间传送控制权。因此, 产生了第二个缺点——执行时间长, 因为它要用更多时间来执行所需的转移操作。

考虑一条比较复杂的完整机器指令的例子。在第2章中, 我们使用了此类指令

Add src, Rdst

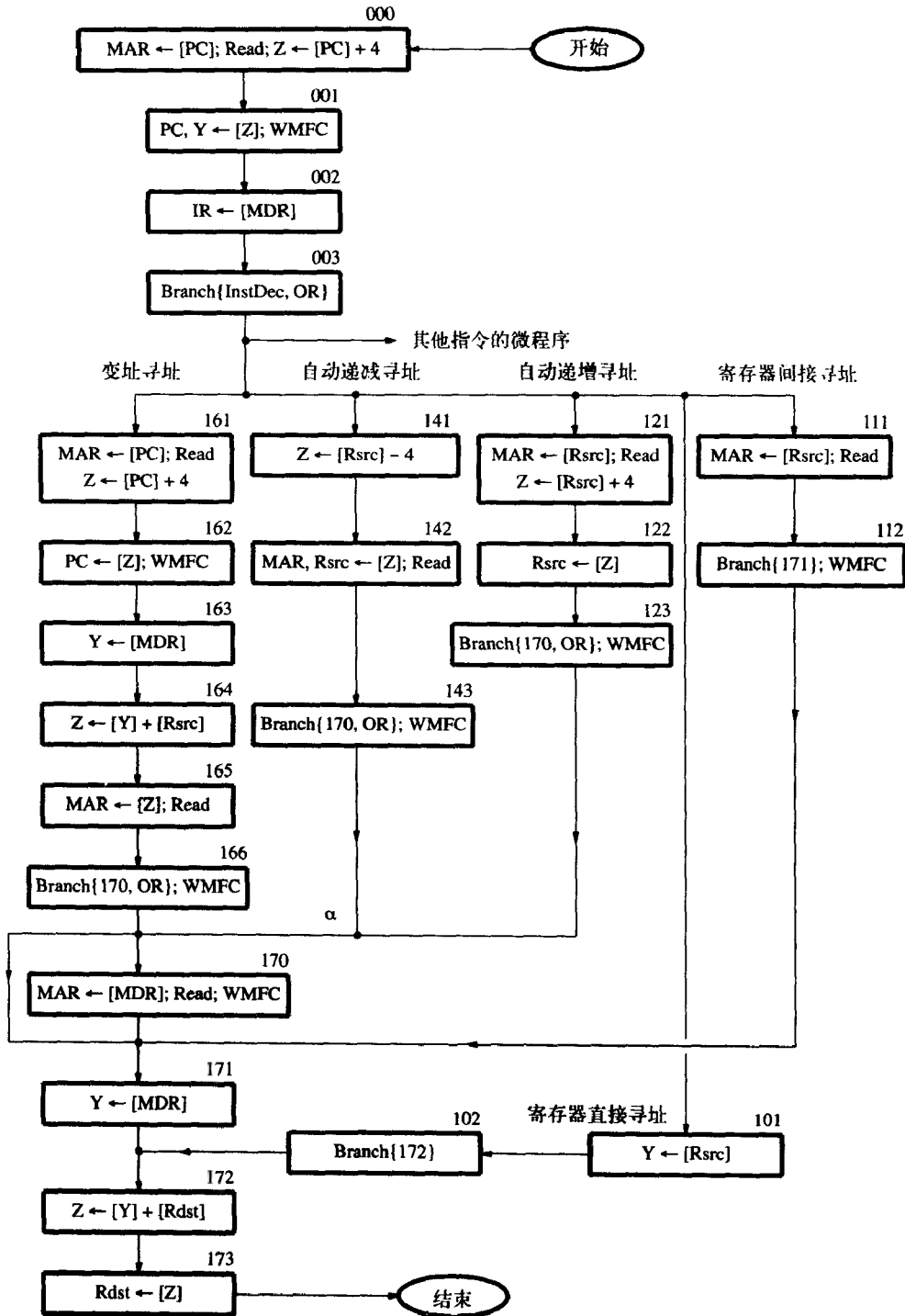
它将源操作数与寄存器Rdst的内容相加并把结果存放在目标寄存器Rdst中。假设源操作数可用下列寻址方式指定: 寄存器、自动递增、自动递减和变址寻址方式, 以及这四种方式的间接形式。我们现在将该指令与图7-1中的处理器结构结合来说明微程序的一种可能实现方法。

**435** 为了易于理解, 在图7-20中以流程图形式给出一个相应的微程序。图中每一个方框对应一个控制方框图中所指示的传送和操作的微指令。微指令位于方框图右上角的八进制数指定的地址单元中。每个八进制数代表三位。我们在该例中使用八进制记数法作为二进制数的简洁标记法。图中的大部分流程图是无须解释的, 不过某些细节还需要详细说明。我们将首先阐述涉及的问题, 接着具体观察图中的微指令流。

#### 使用按位-或技术修改转移地址

图7-20中的微程序说明在一个转移地址处并不一定要发生转移。这是通过共享公共部分组合成简单微程序的直接效果。考虑图中标为 $\alpha$ 的点, 在这一点上, 必须通过直接和间接寻址方式来选择所需要的动作。如果在指令中指定间接方式, 那么就执行170单元处的微指令, 从存储器中取操作数。如果指定为直接方式, 那么必须绕过这个取操作, 而直接转移到171单元处。绕过170单元处微指令的最有效方法是让前一条转移微指令指向地址170, 接着如果包含直接寻址方式, 用一个或门将该地址中最低位改成1。这就是用来修改转移地址的按位-或 (bit-ORing) 技术。

按位-或的一种替代方法是在单元123、143和166处使用两个条件转移微指令。另一种可能方式是在转移微指令中包含两个以下地址的字段, 一个用于直接寻址方式, 一个用于间接寻址方式。这两种方法都不如按位-或技术。



436

图7-20 Add src, Rdst指令对应的微程序流程图

7.5.3 宽转移寻址方式

图7-20中在单元003处的微指令中包括一个宽转移。指令译码器（图中简写为InstDec）生成

刚被取入IR中的指令的微程序首地址。在我们的例子中，寄存器IR保存着加法指令，指令译码器为其产生微指令地址101。但是，该地址不是像现在这样被装入到微程序计数器中。

加法指令的源操作数可以采用多种寻址方式来指定。图中列出了加法指令可能跟随的五种转移。从左到右分别是变址、自动递减、自动递增、寄存器直接和寄存器间接寻址。上面描述的按位-或技术可用在该点修改由指令译码器生成的首地址，使其到达正确的路径上。对于图中所列出的地址，按位-或操作将根据指令中用的寻址方式，将地址101改为五种可能地址值161、141、121、101或111中的一种。

### WMFC的使用

假设在转移微指令中有可能发出一条等待MFC的命令。例如，在单元112处的微指令就是这样做的，它生成一条转向单元171处微指令的转移指令。将这两种操作绑定在一起会带来一个小问题。WMFC信号意味着完成微指令可能要用几个时钟周期。如果允许在第一个时钟周期发生转移，那么就会过早地取出并执行单元171处的微指令。为了避免出现这种情况，必须在存储器传送过程结束之后才可以进行转移，也就是说在等待期间，WMFC信号必须禁止微程序计数器的内容发生任何变化。

### 具体的分析

让我们仔细看一下图7-20的流程图中的一条路径。考虑以自动递增寻址方式访问源操作数的情况。该条路径需要执行指令

437

Add (Rsrc) +, Rdst

其中Rsrc和Rdst是机器中的通用寄存器。图7-21列出了读取和执行该指令的完整微程序。假设指令采用3位字段来表示源操作数的寻址方式，如图所示。位于第10位和第9位的位模式11、10、01和00分别表示变址、自动递减、自动递增和寄存器四种寻址方式。对于每一种方式，都是用第8位来表示它们相应的间接形式。例如，在模式字段中的010表示自动递增方式中的直接形式，而011表示它的间接形式。还假设处理器拥有可以用于寻址目的的16个寄存器，每个寄存器用一个4位代码表示。因此，源操作数可以用模式字段和第7位到第4位表示的寄存器完整描述出来。目标操作数放在第3位到第0位指定的寄存器中。

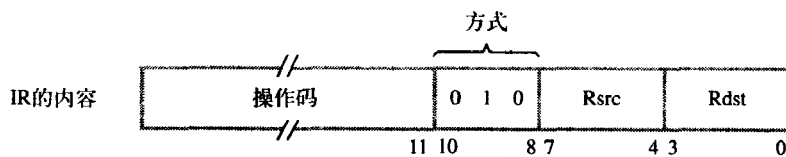
由于16个通用寄存器中的任何一个可能用作判定源和目标操作数的位置，所以微指令分别把各自的控制信号记作Rsrc<sub>out</sub>、Rsrc<sub>in</sub>、Rdst<sub>out</sub>和Rdst<sub>in</sub>。这些信号必须由连接到IR的Rsrc和Rdst地址字段的译码器电路转换成专用寄存器的传输信号。这意味着存在两个译码阶段。第一个阶段将微指令字段进行译码来判定是否包括Rsrc和Rdst寄存器。然后，译码的输出用来将IR中的Rsrc或Rdst字段的内容传送到第二个译码器中，产生寄存器R0到R15的控制信号。

图7-20中的微程序是通过将模式字段中的所有可能值进行组合而得到的，结果是生成一个需要的多转移点结构。在图7-21的实例有两个转移点，从而需要两条转移微指令。在每一种情形下，括号中的表达式指示将要装入μPC中的转移地址，以及该地址如何用按位-或技术进行修改。我们用单元123处的微指令作为分析的例子。在未修改形式中它将转移到170单元处的微指令中，这会再次使用相应的间接寻址方式，并从主存中进行读取。对于直接寻址方式，可以通过将src地址字段（IR中的第8位）中的间接位求反后，与μPC的第0位进行或操作，这样就可以绕过这一步的取指操作。

使用按位-或的另一个例子是单元003处的微指令。实现Add指令的微程序有5个首地址，根据寻址方式来指定源操作数。这些八进制地址中只有中间一位不同。因此，使用按位-或操作来

修改从译码器中获得的模式101的八进制数中间值来实现所需要的转移。由连接到src地址的模式字段（IR的第8、9、10位）的译码器电路提供与该数进行或操作的3位数。由于已经选择了微指令地址，所以这种修改很容易实现； $\mu PC$ 的第4和第5位直接由IR的第9和第10位进行设置。除了一种寻址方式以外，这种方式足以为其他src的寻址方式选择正确的微指令。当 $[\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]$ 等于1时，通过将 $\mu PC$ 的第3位置1来屏蔽寄存器的间接方式。寄存器间接方式是一个特殊情形，因为它是不使用170单元的微指令中的惟一一种间接方式。

438



地址 (八进制)	微指令
000	$PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
001	$Z_{out}, PC_{in}, Y_{in}, WMFC$
002	$MDR_{out}, IR_{in}$
003	$\mu Branch \{ \mu PC \leftarrow 101 \text{ (从指令译码器)}; \\ \mu PC_{5,4} \leftarrow [IR_{10,9}]; \mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8] \}$
121	$Rsrc_{out}, MAR_{in}, Read, Select4, Add, Z_{in}$
122	$Z_{out}, Rsrc_{in}$
123	$\mu Branch \{ \mu PC \leftarrow 170; \mu PC_0 \leftarrow [\overline{IR_8}] \}, WMFC$
170	$MDR_{out}, MAR_{in}, Read, WMFC$
171	$MDR_{out}, Y_{in}$
172	$Rdst_{out}, SelectY, Add, Z_{in}$
173	$Z_{out}, Rdst_{in}, End$

图7-21 Add (Rsrc) +, Rdst的微指令

439

注：对于这种寻址方式，不执行单元170处的微指令。

### 7.5.4 带有下一地址字段的微指令

图7-20中的微程序需要多条转移微指令,这些微指令不执行数据通路中的有效操作;它们只是用来确定下一条微指令的地址,因此降低了计算机的操作速度。当其他微指令被考虑进来时这种情形会变得更糟。如果增加转移微指令,往往会对通常连续执行的所有微指令分配连续地址的能力产生限制。

该问题促使我们重新评估建立在可递增的 $\mu PC$ 基础上的顺序技术。一种有效的替代方法是在每条微指令中增加一个地址字段来指示要提取的下一条微指令的位置。实际上,这意味着每条微指令除了具有其他功能以外,还是一条转移微指令。

这种灵活性是通过地址字段增加附加的位而得到的。这种损失的严重性评估如下:在典型计算机中,用不超过4K的微指令来设计一个完整微程序,其中每条微指令大约占50到80位。这表示要求有12位地址字段。因此,大约需要六分之一的控制存储器容量用于寻址。即使需要

扩展微程序，也只能将地址字段稍微有所增大。

这种方法的最显著优点是几乎省去了单独的转移微指令。而且，微指令的地址分配也几乎不受限制。这些优点大大抵消了它的负面影响，也使得该技术很具有吸引力。由于每条指令包含下一条指令的地址，所以不需要计数器来记录顺序地址。因此， $\mu PC$ 用微指令地址寄存器( $\mu AR$ )来代替，将每条微指令的下一个地址字段域装入 $\mu AR$ 。具有该特性并且支持按位-或操作的新控制结构如图7-22所示。下一个地址位经过或门传送给 $\mu AR$ ，以便地址可在IR中的数据、外部输入和条件码的基础上进行修改。译码器电路在IR中操作码的基础上产生指定微程序的起始地址。

现在运用图7-22的微程序控制结构重新考虑图7-21的例子。我们需要几条控制信号，这些控制信号不包含在图7-19的微指令格式中。用名称Rsrc和Rdst作为对寄存器R0到R15描述的替换，它们可根据IR中src和dst字段的数据译成实际控制信号。带有按位-或技术的转移要求在微指令中包含适当的命令。在图7-20的流程图中，003单元的微指令中根据源操作数的寻址方式使用按位-或的方法来判断下一条微指令的地址。寻址方式由指令寄存器中的第8位到第10位表示，如图7-21所示。信号 $OR_{mode}$ 用来控制是否使用按位-或技术。在微指令123、143和166中，采用按位-或来判断源操作数是否使用间接寻址。信号 $OR_{indsrc}$ 就是用于此目的的。为了简单起见，我们利用微指令中单独的位来表示这些信号。利用微指令中一位来表示指令译码器的输出何时传送到 $\mu AR$ 中。最后，每条微指令中包含下一条微指令地址的8位字段。图7-23是这类微指令的一个完整格式，这个格式是对图7-19中格式的扩充。

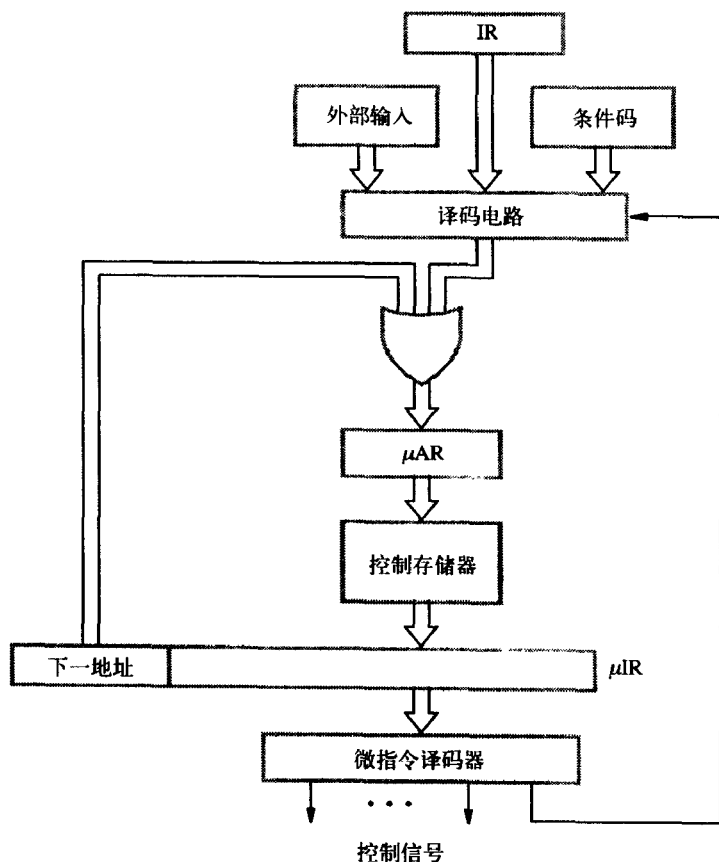


图7-22 微指令顺序结构

我们可以采用这类微指令来实现图7-21的微程序，如图7-24所示。修改后的程序将少一条微指令。因为我们将单元123处的转移微指令与它的前一条微指令合并。当微指令时序由 $\mu$ PC控制时，End信号用于 $\mu$ PC复位，使它指向取出并要执行的下一条机器指令微指令的首地址。在我们的例子中，这个首地址是000<sub>8</sub>。不过图7-24中的微程序并不会因为产生End信号而终止。在这类结构中，一般起始地址并不是采用由End信号触发的复位机制来指定的，相反是采用显式方式在F0字段中指定的。

441

微指令

F0	F1	F2	F3
F0 (8位)	F1 (3位)	F2 (3位)	F3 (3位)
下一微指令地址	000: No transfer 001: PC <sub>out</sub> 010: MDR <sub>out</sub> 011: Z <sub>out</sub> 100: Rsrc <sub>out</sub> 101: Rdst <sub>out</sub> 110: TEMP <sub>out</sub>	000: No transfer 001: PC <sub>in</sub> 010: IR <sub>in</sub> 011: Z <sub>in</sub> 100: Rsrc <sub>in</sub> 101: Rdst <sub>in</sub>	000: No transfer 001: MAR <sub>in</sub> 010: MDR <sub>in</sub> 011: TEMP <sub>in</sub> 100: Y <sub>in</sub>

F4	F5	F6	F7
F4 (4位)	F5 (2位)	F6 (1位)	F7 (1位)
0000: Add 0001: Sub ⋮ 1111: XOR	00: No action 01: Read 10: Write	0: SelectY 1: Select4	0: No action 1: WMFC

F8	F9	F10
F8 (1位)	F9 (1位)	F10 (1位)
0: NextAdrs 1: InstDec	0: No action 1: OR <sub>mode</sub>	0: No action 1: OR <sub>indsrc</sub>

442

图7-23 7.5.3节中实例的微指令格式

图7-25中给出了图7-22控制结构的更加具体的图示。它给出了控制信号如何从微指令字段中译出以及如何在控制序列中使用。在图7-26中给出了按位-或的具体电路。

7.5.5 预取微指令

微程序控制的一个缺点是它导致了操作速度的降低，因为从控制存储器读取微指令要花费一些时间。如果在当前微指令执行的同时预取下一条微指令就可以获得较快的操作速度。在这种方法中，执行时间与取指令时间是重叠的。



八进制地址	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
000	00000001	001011	0010000	01	0000	01	1	0	0	0	0
001	00000010	011001	1000000	00	0000	00	0	1	0	0	0
002	00000011	010010	0000000	0000	0000	00	0	0	0	0	0
003	00000000	000000	0000000	0000	0000	00	0	0	1	1	0
121	01010010	100011	0010000	01	0000	01	1	0	0	0	0
122	01111000	011100	0000000	0000	0000	00	0	1	0	0	1
170	01111001	010000	0010000	01	0000	01	0	1	0	0	0
171	01111010	010000	1000000	0000	0000	00	0	0	0	0	0
172	01111011	101011	0000000	0000	0000	00	0	0	0	0	0
173	00000000	011101	0000000	0000	0000	00	0	0	0	0	0

图7-24 采用下一微指令地址字段实现（编码信号参见图7-23）图7-21中的微程序

443

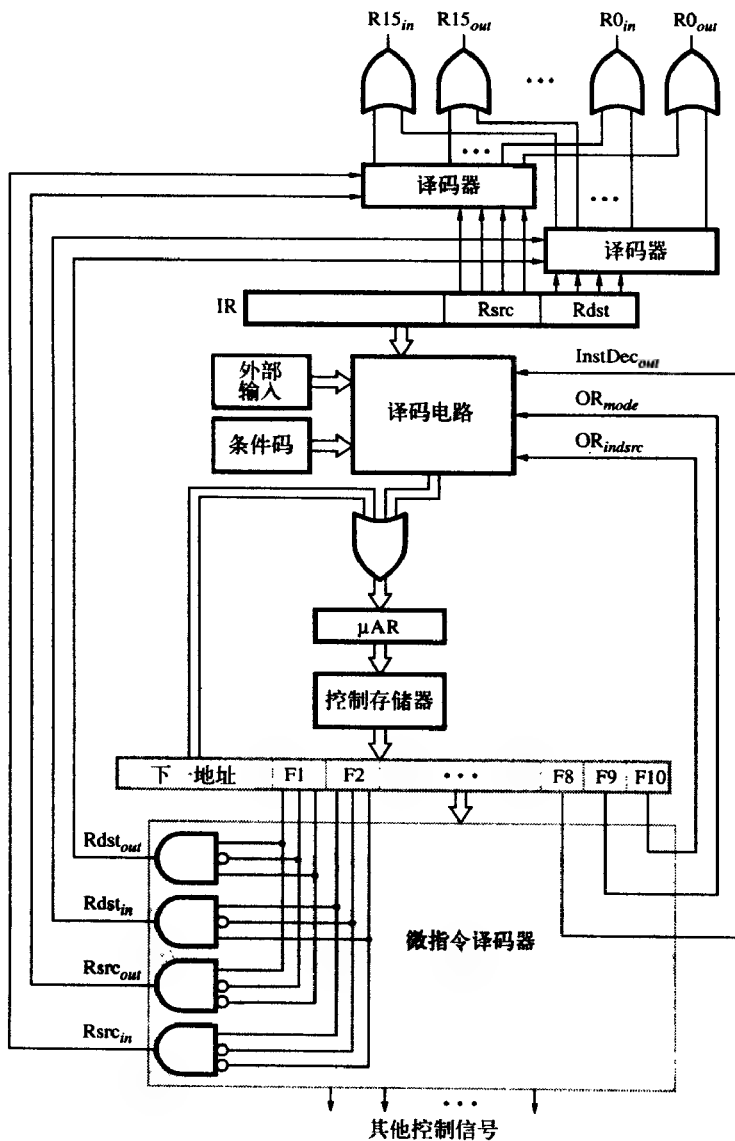


图7-25 控制信号生成电路的一些具体细节

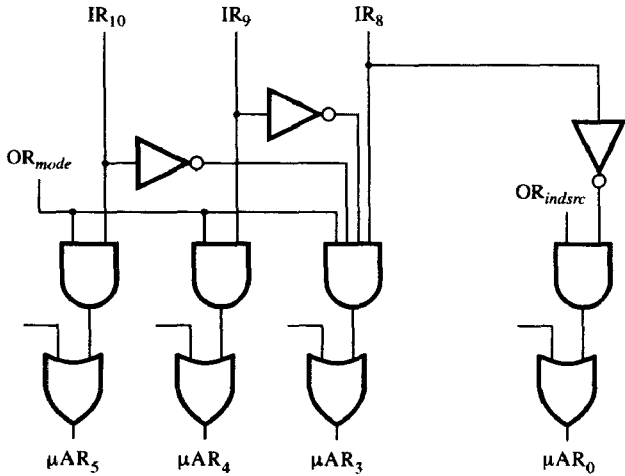


图7-26 按位-或控制电路（图7-25中译码电路的一部分）

预取微指令为系统的结构带来一些困难。有时，状态标志位和当前执行的微指令的操作结果要用来判定下一条微指令的地址。因此，直接预取偶尔会发生取错微指令的情况。这时，必须按照正确地址重取，这就需要有比较复杂的硬件结构。虽然这种技术存在一些微小的瑕疵，不过对于这种预取技术人们还是认可的，并经常会采用它。

7.5.6 仿真

微程序控制的主要功能是为机器指令的执行提供一种简单、灵活、相对便宜的方法。然而它也带来了其他的好处。它在使用机器资源方面具有一定的灵活性，所以可以实现各类指令。在一台给定的具有某种指令集的计算机上，有可能去定义其他的机器指令并利用附加的微程序来实现这些机器指令。

444

对前面的想法进行扩展可以带来另一种有趣的可能性。假设我们为给定计算机M1增加一个全新的指令集，实际上该指令集是另一台计算机M2的指令集。采用M2的机器语言编写的程序可在计算机M1上运行，即M1仿真M2。仿真使我们能够用较新的机器来代替过时的机器。如果用作替代的计算机能够完全地仿真原计算机，那么对于现存的程序不必做任何软件上的更改便可以运行。因此，仿真技术可以使得计算机系统的过渡变得容易，并且改动较少。

当所涉及的机器具有相似的体系结构时，仿真就更加容易。不过，对于完全不同的体系结构，仿真也是可以实现的。

7.6 结束语

在本章中，我们全面介绍了计算机中央处理器的结构。商用机所使用的结构是我们所提出结构的各种变种。对于一种具体结构的选择往往涉及到执行速度与实现成本之间的权衡。其他因素如采用的技术、修改的灵活性以及计算机指令集需要的特殊功能等，也起到一定的作用。

在这里还介绍了实现处理器控制部件的两种方法——硬件控制和微程序控制。当所关心的是操作速度时，硬件控制是最好的方法。而在实现指令集方面，微程序控制提供了相当大的灵活性。

445

## 习题

- 7.1 当从主存储器读出或向主存储器写入的时候,为什么需要等待存储器功能完成(WMFC)这一步?
- 7.2 一个处理器使用类似于图7-6的控制序列。假设存储器读或写操作与一个内部处理器操作占用相同的时间,并且处理器和存储器由同一时钟控制。估算一下这个序列的执行时间。
- 7.3 对于一台存储器访问时间等于处理器时钟周期两倍的机器,重新回答7.2的问题。
- 7.4 假设沿总线传输的时间延迟和经过图7-1中ALU的时间延迟分别是0.3ns和2ns。寄存器建立的时间是0.2ns,保持时间是0。所需要的最小时钟周期是多少?
- 7.5 为下列每条指令写出图7-1的总线结构所要求的控制步:

- (a) 将(立即)数NUM与寄存器R1相加。
- (b) 将存储器单元NUM处的内容与寄存器R1相加。
- (c) 将地址在存储器单元NUM中的内容与寄存器R1相加。

假设每条指令由两个字构成。第一个字表明操作和寻址方式,第二个字包含数NUM。

- 7.6 在习题7.5的三条指令中有许多通用的控制步。然而,这些控制步中有一些发生在控制步计数器的不同计数处。给出一种利用这些步来降低图7-11的编码器块复杂度的方法。
- 7.7 考虑具有图7-6给出的控制序列的加法指令。处理器被连续运行的时钟所驱动,以便在一个持续期间里每一个控制步都是2ns。假设存储器的读操作需要占用16ns,那么,处理器在第2步和第5步要等待多久?在这条指令执行期间,处理器空闲时间占多少百分比?
- 446 7.8 一台32位的按字节可寻址的机器,其寻址方式包括自动递增和自动递减方式。在这两种方式中,根据操作数的长度,地址寄存器的内容分别增加或减少1、2或4。对图7-1进行一些修改以简化这种操作。
- 7.9 对于在图7-1的处理器上执行的指令

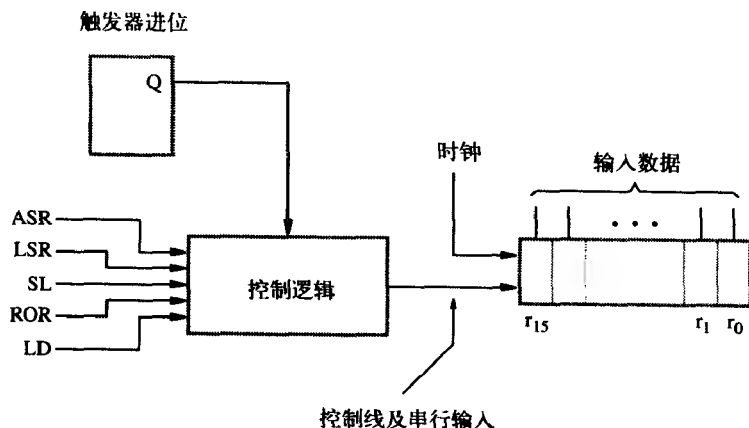
MUL R1, R2

给出一种可行的控制序列。这条指令将寄存器R1和R2的内容相乘,将结果存在R2中。在乘积中如果还有高位部分,就丢弃。根据建议补充控制信号,并假设多路复用器的结构如图6-7所示。

- 7.10 对于一个包含图7-8给出结构的处理器,写出负数转移(Branch-on-Negative)指令的控制步。
- 7.11 写出用来执行第3章中描述的转移到子程序(Branch-to-Subroutine)指令的控制步。假设处理器拥有图7-1的内部结构。
- 7.12 对于图7-8中的处理器,重新回答7.11的问题。
- 7.13 图7-3是用于实现处理器寄存器边沿触发的触发器。考虑将数据从一个寄存器传送到另一个寄存器的操作。如果边沿触发器被诸如图A-27中的简单门控锁存器代替,仔细分析该操作的时序并解释可能遇到的任何潜在困难。
- 7.14 图7-3中的多路复用器和反馈连接消除了对时钟输入进行控制的需求,它是作为寄存器输入的一种使能方式。运用时序图解释可能发生的问题,假设采用时钟控制。
- 7.15 假设图7-8中的寄存器文件用作RAM。在任何一个给定的时间里,该RAM中的位置可由读或写操作访问。在操作 $R1 \leftarrow [R1] + [R2]$ 期间,寄存器R1既是源寄存器又是目标寄存器。阐述在主从模式中如何在RAM的输入端或输出端运用新添的锁存器来操作文件。利用时序

图解释你的新设计如何使寄存器R1在同一时钟周期中既用作源寄存器又用作目标寄存器。

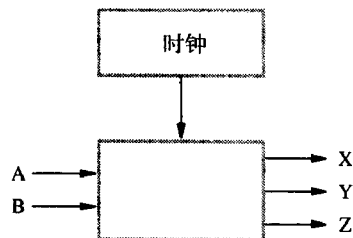
- 7.16 在图7-11中将Run信号置为0,以阻止在等待存储器读或写操作完成时控制步计数器增加。查看图7-5中的时序图,画出产生该信号的控制电路状态图。设计一个合适的电路。
- 7.17  $MDR_{inE}$  控制信号随着在一个时钟周期中控制信号Read被确定而确定,而当存储器传送完成后被取消,如图7-5所示。设计一个合适的电路生成 $MDR_{inE}$ 。
- 7.18 考虑一台含有图7-1中结构的16位可按字节寻址的机器。偶地址和奇地址的字节分别在存储器总线的高8位和低8位上传送。给出一种合适的控制方法,将寄存器MDR连接到存储器总线 447 和内部处理器总线上允许进行字节传送。当保存字节时,它应总是在处理器内部的低位字节上。
- 7.19 利用反相器和延迟元件设计一个振荡器。假设延迟元件引入延迟 $T$ ,振荡频率是多少?对振荡器进行修改,使其可以在同步输入RUN的控制下启动和停止。当振荡器被停止时,输出的最后一个脉冲宽度必须等于 $T$ ,该时间是使RUN变为非活动状态的一个独立时间。
- 7.20 处理器中一些控制步的完成比其他的控制步占用的时间长。希望产生一个由Long/Short信号控制的时钟信号,使得当该信号等于1时控制步的持续时间是它的两倍长度。假设控制步计数器有一个允许输入端,并且计数器在时钟上升沿如果Enable=1时递增。设计一个产生Enable信号的电路以便根据需要改变控制步的大小。
- 7.21 将移位寄存器的输出反相后反馈到它的输入端,以形成一个叫做Johnson计数器的计数电路。  
(a) 4位Johnson计数器的计数序列是什么,从状态0000开始?  
(b) 说明如何运用Johnson计数器来产生图7-11中的定时信号 $T_1$ 、 $T_2$ 等,假设最大的时间间隔为10。
- 7.22 一个处理器的ALU使用如图P7-1所示的移位寄存器来实现移位和循环移位操作。该寄存器的控制逻辑输入包括:
- ASR 算术右移
  - LSR 逻辑右移
  - SL 左移
  - ROR 循环右移
  - LD 并行装入



图P7-1 习题7.22中移位寄存器的控制结构

所有的移位和取操作由一个时钟输入控制。移位寄存器由边沿触发D触发器实现。为控制逻辑和移位寄存器的 $r_0$ 、 $r_1$ 和 $r_{15}$ 位画出一个完整的逻辑图。

- 7.23 图P7-2的数字控制器有三个输出X、Y和Z以及两个输入A和B。它由外部时钟驱动。控制器连续经历下列事件：在第一个时钟周期的开始，X线置为1。在第二个时钟周期的开始，Y线或Z线置1，这分别取决于在前一时钟周期里A线等于1还是0。接着控制器等待直到B线置为1。在下一个时钟的上升沿，控制器置输出Z为1，持续时间为1个时钟周期。接着将所有输出信号复位为0，持续时间为1个时钟周期。该序列是重复的，从下一时钟的上升沿开始。为这个控制器画出状态图并给出一个合适的逻辑设计。



图P7-2 习题7.23中的数字控制器

- 7.24 为指令

MOV X (Rsrc), Rdst

写一个像图7-21所示的微程序，其中源和目标操作数分别以变址和寄存器寻址方式指定。

- 7.25 BGT (Branch if >0) 机器指令以表达式 $Z + (N \oplus V) = 0$ 作为它的转移条件，其中Z、N和V分别是零、负数、溢出条件标志。写出一个能够实现这条指令的微程序。画出用来测试条件码的电路。
- 7.26 写一个能实现BGT（如果大于零转移）、BPL（如果为正转移）和BR（无条件转移）指令的联合微程序。BGT和BPL指令的转移条件分别是 $Z + (N \oplus V) = 0$ 和 $N = 0$ 。所需要的微指令总数是多少？如果每条机器指令使用一个单独的微程序，需要多少条微指令？
- 7.27 图7-21是一个微程序实例，其中按位-或用于修改微指令地址。写一个不使用按位-或而使用条件转移微指令的等价程序。需要补充多少微指令？假设条件转移微指令可用于测试IR中的一些位。

449

- 7.28 指出图7-20的微程序应如何修改以执行68000微处理器指令

ADD src, Rdst

- 7.29 说明图7-20的流程图应如何修改以实现一般指令

MOVE src, dst

其中源和目标都可以使用所示的五种寻址方式中的任何一种。

- 7.30 图7-3给出了对应于微程序计算机中机器指令的微指令序列的一部分。微指令B后是C、E、F或I，这取决于机器指令寄存器的 $b_6$ 和 $b_5$ 位。比较下列描述的三种可能实现方式。

(a) 微指令序列通过微程序计数器实现。由微指令形式

If  $b_6b_5$  branch to X

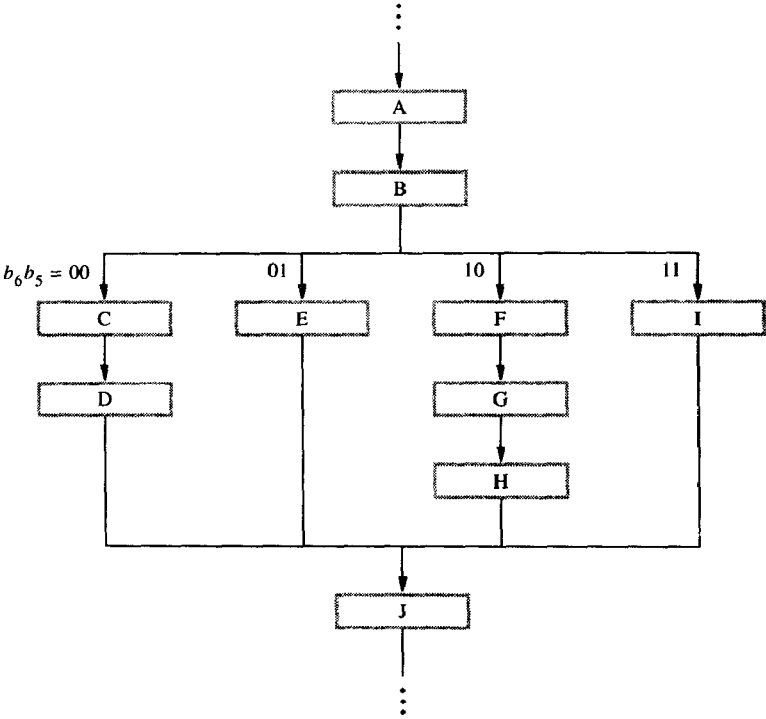
实现转移，其中 $b_6b_5$ 是转移条件，X是转移地址。

(b) 与a部分相同，除了转移微指令的形式是

Branch to X, OR

其中X是基本转移地址。转移地址由 $b_5$ 和 $b_6$ 位以及X中的相应位按位-或进行修改。

450



图P7-3 习题7.30用到的微指令序列方式

(c) 每条微指令中的一个字段表明下一条微指令的地址，它含有按位-或的性能。  
对于a到c部分的每一种实现，为图P7-3中的所有微指令分配合适的地址。注意在一些情形下你可能需要插入转移指令。可以选择任意地址，只要它们与所使用的序列方法一致。例如，在a部分你可以选择地址如下：

地 址	微 指 令
00010	A
00011	B
00100	If $b_6b_5 = 00$ branch to XXXXX
...	...
XXXXX	C

- 7.31 我们希望减少图7-19中控制信号编码所需要的位数。推荐一种新编码，使位数减少两位。新编码将如何影响实现一条指令所需要的控制步数量？
- 7.32 为图7-19中控制信号推荐一种新编码，使一条微指令所需要的位数减少到12。指出新编码对图7-6和图7-7中控制序列的影响。
- 7.33 为微指令推荐一种格式，类似于图7-19，假定处理器的结构如图7-8所示。
- 7.34 横向和纵向微指令格式的相对优点是什么？将你的答案与习题7.31和习题7.32的答案联系起来考虑。
- 7.35 硬件控制和微程序控制的优点和缺点是什么？



### 本章目标

在本章中你将学习以下内容:

- 并行执行机器指令的流水线方式
- 引起流水线处理器性能下降的各种阻塞以及降低阻塞影响的方法
- 流水线中的硬件和软件含义
- 流水线对指令集设计的影响
- 超标量处理器

453

在前面的章节中介绍了计算机的基本构件块。在本章中，我们详细讨论在现代计算机中用来获取更高性能的流水线技术。首先阐述流水线的基础以及它是如何改善性能的。然后验证便于流水线执行的机器指令特征，并证明指令和指令序列的选择对性能有重大影响。流水线结构需要娴熟的编译技术，而用于这一目标的优化编译器已开发出来。相对于其他编译器，这种编译器重新安排操作顺序以使流水线尽可能发挥出最大作用。

### 8.1 基本概念

程序执行的速度受到许多因素影响。提高性能的一种方法是利用快速电路技术来设计处理器和主存储器，另一种可能是对硬件进行调整，使其能在同一时间里执行多项操作。采用后一种方法，即使用到的每一种操作所需的执行时间都不改变，每秒执行操作的次数也会增加。

前面已经多次碰到过并发动作。在第1章中，我们介绍了多道程序的概念，并阐述了如何才能将I/O传输与计算活动同时进行。DMA设备使其成为可能，因为一旦I/O传送由处理器启动后，DMA设备就可以独立地执行I/O传送。

在计算机系统中，流水线是组织并发活动的一个非常有效的方法，它的基本思想很简单。在制造工厂里经常会看到流水线，那里的流水线通常是作为装配线操作的。毫无疑问，读者对汽车制造厂中用到的装配线是很熟悉的。装配线的第一站准备汽车底盘，第二站安装车身，下一站安装发动机等等。当一些工人在一辆汽车上安装发动机时，另一些人在另一辆汽车的底盘上安装车身，还有一些人为第三辆车准备新的底盘。虽然在一辆给定的车上完成这些工作可能会花费几天时间，但是每隔几分钟会有一辆新车从装配线的末端开出来。

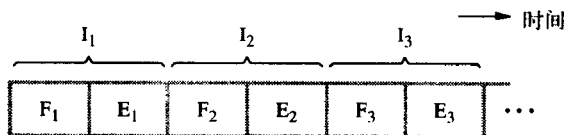
考虑一下流水线的思想如何能用于计算机中。处理器通过将指令逐个取出并完成执行来处理一道程序。设 $F_i$ 和 $E_i$ 表示指令 $I_i$ 的取指令与执行指令阶段。一道程序的执行过程中包含一系列



取指令和执行指令的阶段,如图8-1a所示。

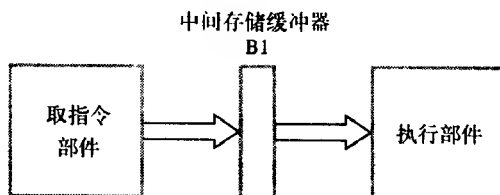
454

现在假设一台计算机有两个独立的硬部件,一个用来取指令,另一个用来执行指令,如图8-1b所示。从取指令部件取的指令存入中间存储缓冲器B1中。使用缓冲器的目的是为了使得执行部件在执行当前指令的同时,取指令部件可以取下一条指令。操作的结果保存到由指令指定的目标位置中。为了便于讨论,我们假设指令操作的源操作数和目标操作数都在标为“执行部件”的模块内部。

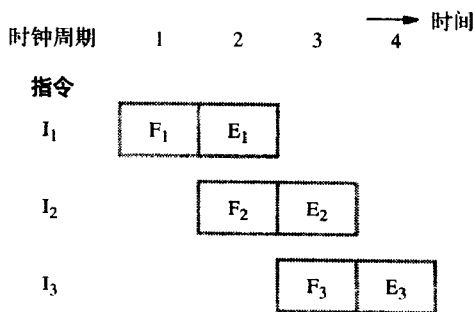


a) 顺序执行

计算机是由时钟控制的,任何一个指令的读取和执行阶段都能够在一个时钟周期内完成。计算机的操作过程如图8-1c。在第一个时钟周期,取指令部件读取指令 $I_1$ (步骤 $F_1$ )并在时钟周期末将它存储在缓冲器B1中。在第二个时钟周期,取指令部件进行指令 $I_2$ (步骤 $F_2$ )的取指令操作。同时,执行部件执行由指令 $I_1$ 指定的操作。指令 $I_1$ 可以从缓冲器B1中(步骤 $E_1$ )得到。到第二个时钟周期末,指令 $I_1$ 的执行结束并且指令 $I_2$ 已经取出。指令 $I_2$ 取代不再需要的 $I_1$ 存储在B1中,由执行部件在第三个时钟周期执行步骤 $E_2$ ,同时取指令部件取指令 $I_3$ 。在这种方式下,取指令部件和执行部件都一直处于工作状态。如果图8-1c中的方式能维持很长时间,那么指令执行的完成率会是图8-1a描述的顺序操作的两倍。



b) 硬件结构



c) 流水线执行

455

总之,图8-1b中的取指令部件和执行部件构成2段流水线。在流水线中,每一段实现指令处理中的一步。中间存储缓冲器B1用来存储从一段传递到另一段的信息。在每一个时钟周期末,新的信息装入这个缓冲器。

图8-1 指令流水线的基本思想

指令的处理并不是必须只分成两步。例如,流水线处理器可以采用四步来处理指令,如下:

- F 取指: 从存储器中读指令
- D 译码: 指令译码并取源操作数
- E 执行: 执行指令指定的操作
- W 写: 将结果保存到目标位置

这种情况的事件顺序如图8-2a所示。在任何时刻都有四条指令在执行。这意味着需要四个不同的硬部件,如图8-2b所示。这些部件必须能够同时执行它们的任务并且相互之间没有干扰。信息通过存储缓冲器从一个部件传递到另一个部件。当指令通过流水线时,下游各段所需要的所有信息必须跟着传递。例如,在第4个时钟周期,缓冲器中的信息如下:

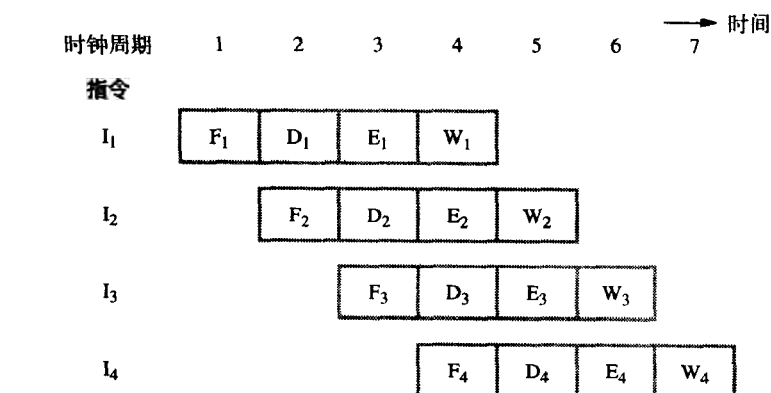
- 缓冲器B1存有指令 $I_3$ ,它在第3个周期取出并且正由指令译码部件进行译码。
- 缓冲器B2存有指令 $I_2$ 的源操作数以及要执行的操作的说明。该信息由译码硬件在第3个周

期产生。B2还存有指令 $I_2$ 的写阶段（步骤 $W_2$ ）所需要的信息。即使E段不需要该信息，该信息也必须在随后的时钟周期传递给W段，以使该段能执行所要求的写操作。

- 缓冲器B3存有执行部件产生的结果以及指令 $I_1$ 的目标信息。

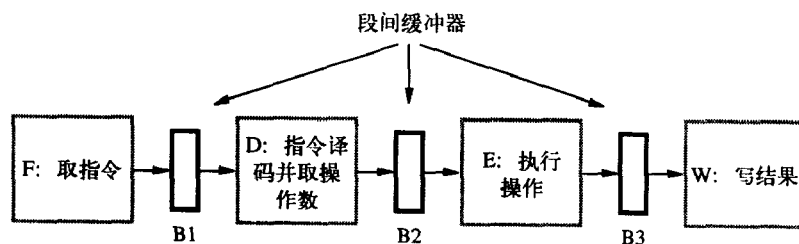
### 8.1.1 高速缓存的作用

我们希望流水线的每一段都能在一个时钟周期内完成它的操作。因此，时钟周期应该足够长以便在任一阶段上执行的任务都可以完成。如果不同的部件需要的时间不同，那么时钟周期必须以最长的任务完成时间为准。提前完成任务的部件在余下的时钟周期中处于空闲状态。所以，如果不同段的任务所需的时间大致相同，那么流水线对性能的提高会非常有效。



456

a) 分成四步的指令执行



b) 硬件结构

图8-2 一条4段流水线

这种考虑对于取指令阶段尤其重要。在图8-2a中取指令阶段安排了一个时钟周期，时钟周期必须等于或大于完成取操作所需要的时间。然而，主存储器的访问时间可能是处理器内部执行基本流水线段操作时间的十倍，例如两个数相加。因此，如果每条指令都要从主存储器中读取，流水线就没有什么价值了。

高速缓冲存储器（高速缓存）的使用解决了存储器访问的问题。尤其当高速缓存与处理器在同一芯片上时，高速缓存的访问时间通常与执行处理器内其他基本操作所需要的时间相同。这就可以将指令的读取和处理分成持续时间基本相等的几步。每一步由不同的流水段执行，并且时钟周期的选择与持续时间最长的一步相对应。

457

## 8.1.2 流水线性能

在图8-2中,流水线处理器在每一个时钟周期内完成一条指令的处理,这意味着指令处理的速率是顺序操作的四倍。流水线所带来的性能的提高与流水阶段数成正比。不过,这种递增只有在如图8-2a所示的流水线操作在整个程序执行期间能够一直保持不间断的情况下才能获得。遗憾的是,实际中并不是这样的。

由于很多原因,对于给定的指令,流水线中的某个段可能无法在分配的时间段内完成它的任务。例如,在图8-2b中4段流水线的E段,它负责算术和逻辑操作,分配一个时钟周期完成这项任务。虽然对于大多数操作这个时间可能是足够的,但是对于某些操作,例如除法,可能需要较长的时间才能完成。图8-3给出了一个例子,在这个例子中指令 $I_2$ 指定的操作需3个周期来完成,即从第4个周期到第6个周期。因此,在第5和第6周期,因为没有要处理的数据,所以写阶段不能做任何事情。同时,缓冲器B2中的信息必须保持到执行阶段完成。由于B1中的信息不能被覆盖,所以第2段与第1段都无法接受新指令。于是 $D_4$ 和 $F_5$ 必须如图所示的那样向后延迟。

图8-3给出的流水线操作拖延了两个时钟周期。正常的流水线操作在第7个周期重新开始。任何引起流水线拖延的条件称为阻塞。前面我们看到的例子是数据阻塞。数据阻塞是指指令中的源或目标操作数不能在流水线预计的时间里获得。所以有些操作不得不延迟,因此造成流水线操作也被拖延了。

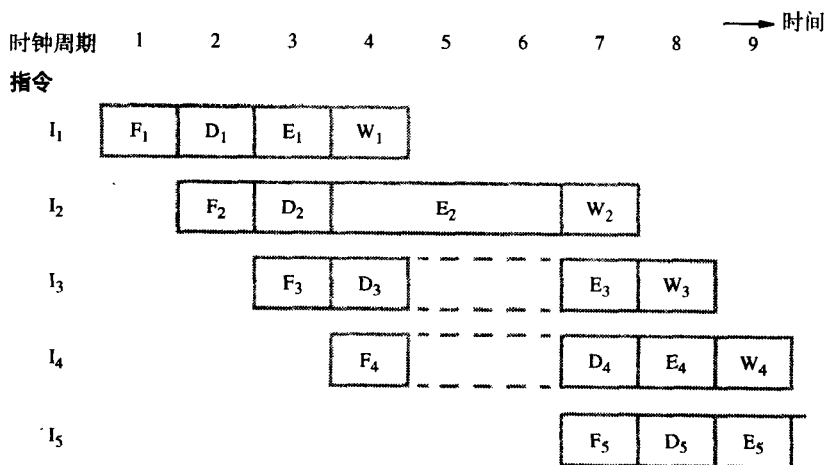
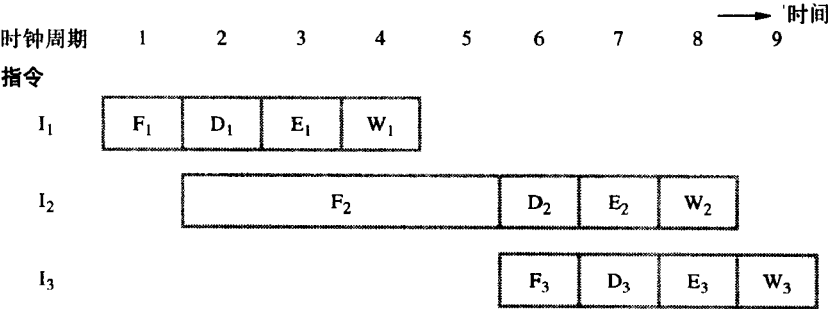


图8-3 执行操作占用多个时钟周期的影响

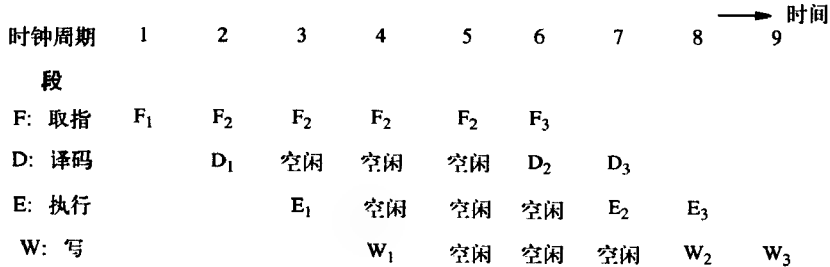
流水线可能也会因为指令的引用延迟而被拖延。例如,可能是因为高速缓存失效(未命中),必须从主存中提取指令而造成的。这种阻塞一般叫做控制阻塞或指令阻塞。流水线操作中的高速缓存失效的影响如图8-4所示。在第一个周期指令 $I_1$ 从高速缓存中取出,并且正常地进行。然而,在第二个周期开始时指令 $I_2$ 的取操作出现了高速缓存失效。取指令部件现在必须挂起任何后续的取指请求,等待 $I_2$ 的出现。假设指令 $I_2$ 在第5个周期末取到并装入缓冲器B1。流水线在那一点重新开始它的正常操作。

在高速缓存失效的情形下流水线操作的另一种表示方法如图8-4b所示。该图给出了在每个时钟周期中每一段流水线实现的功能。注意译码部件在第3个周期到第5个周期处于空闲状态,执行部件在第4个周期到第6个周期是空闲的,写部件在第5个周期到第7个周期是空闲的。这样

的空闲称作拖延。通常也称为流水线中的气泡。一旦在流水线的某段中产生了延迟，气泡就会向下移动直到到达最后一个部件。



a) 在连续的时钟周期下指令的执行步骤



b) 在连续的时钟周期下处理器每段实现的功能

图8-4 F2中高速缓存失效引起的流水线延迟

在流水线操作中可能会遇到的第三类阻塞是结构阻塞。这种情形出现在两条指令同时请求使用某个硬件资源时，它经常在访问存储器时发生。一条指令在完成它的执行阶段或写阶段时可能需要访问存储器，而此时正有另外一条指令在完成取指令。如果指令和数据位于同一个高速缓存单元，那么只允许有一条指令能够进行，另一条指令将被延迟。许多处理器使用独立的指令和数据高速缓存来避免这种延迟。

图8-5给出的是一个结构阻塞的例子。该图说明了load指令

Load X (R1), R2

应当如何适应举例中的4段流水线操作。存储器地址X+[R1]在第4个周期的E<sub>2</sub>步计算，接着在第5个周期访问存储器。在第6个周期将从存储器中读到的操作数写入到寄存器R2中。这表示指令的执行阶段占用两个时钟周期（第4个周期和第5个周期）。它使流水线拖延了一个时钟周期，因为指令I<sub>2</sub>和I<sub>3</sub>在第6个周期都要求访问寄存器文件。即使指令和它使用的数据都已获得，流水线也会因为一个硬件资源（例如寄存器文件）不能同时处理两种操作而产生延迟。如果寄存器文件有两个输入端口，即它允许两个写操作同时进行，流水线就不会被拖延。总之，通过在处理器芯片上提供充足的硬件资源可以避免结构阻塞。

理解流水线并不会因为单指令执行速度的加快而可以提高吞吐量是很重要的。其中吞吐量是通过指令执行完成的速率来度量的。任何时候，流水线的一段不能在一个时钟周期完成它的

操作,就会产生延迟,性能就会下降。因此,在一个时钟周期内执行一条指令实际上是图8-2b所示的流水线处理器可以达到的吞吐量的最大值。

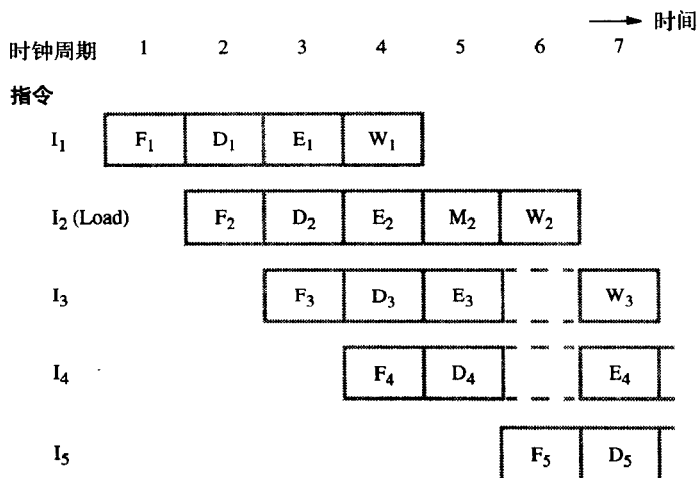


图8-5 Load指令对流水线时序的影响

设计处理器的一个重要目标是识别出可能导致流水线延迟的所有阻塞,并找出减少阻塞影响的方法。在随后的几节中将讨论各种阻塞,首先是数据阻塞,然后是控制阻塞。对于每一种情况会给出一些减缓阻塞负面影响的技术。在8.8节会谈到性能评估的问题。

## 8.2 数据阻塞

数据阻塞是操作数据由于某种原因被延迟而导致流水线拖延的情况,如图8-3中说明的那样。我们现在具体讨论数据可用性的问题。

假设一个程序包含两条指令, I<sub>2</sub>紧跟着I<sub>1</sub>。当这个程序在流水线中执行时, I<sub>2</sub>可以在I<sub>1</sub>的执行结束之前开始执行。这意味着I<sub>2</sub>可能无法使用I<sub>1</sub>产生的结果。我们必须确保用流水线处理器执行指令所得到的结果与同样的指令按照顺序执行所得到的结果是相同的。下面通过一个简单的例子来说明,通过并行执行操作可能会得到一个错误的结果。假设A=5,考虑下列两个操作:

$$A \leftarrow 3 + A$$

$$B \leftarrow 4 \times A$$

当这两个操作按照给定的顺序执行时,结果是B=32。但如果它们并行执行,用来计算B的A值可能就是原值5,这样会导致不正确的结果。如果这两个操作是在同一个程序的指令中完成,那么必须是按顺序地执行这两条指令。因为在第二条指令中用到的数据依赖于第一条指令的结果。相反,以下两个操作

$$A \leftarrow 5 \times C$$

$$B \leftarrow 20 + C$$

能够并行地执行。因为这两个操作之间是相互独立的。

本例说明了必须施加基本约束以确保结果的正确性。当两个操作彼此依赖时,必须以正确的次序顺序执行。这种非常明显的条件对结果有着深远的影响。理解它的含义是理解在流水线

计算机中碰到的各种设计选择和权衡的关键。

考虑图8-2中的流水线。以上描述的数据依赖性在当一条指令的目标操作数用作下一条指令的源操作数时产生。例如，两条指令

Mul R2, R3, R4

Add R5, R4, R6

产生了数据依赖性。乘法指令的结果放入寄存器R4，R4接着又作为加法指令的一个源操作数。假设乘法操作用一个时钟周期来完成，执行的进程如图8-6所示。当译码部件在第3个周期对加法指令译码时，已了解到R4要用作源操作数。因此，直到乘法指令的W步完成后，加法指令的D步才能完成。D<sub>2</sub>步的完成必须延迟到第5个周期，在图中以D<sub>2A</sub>步表示。指令I<sub>3</sub>在第3个周期获得，但是由于D<sub>3</sub>步不能在D<sub>2</sub>步之前，所以I<sub>3</sub>的译码必须延迟。因此，流水线执行产生了两个周期的延迟。

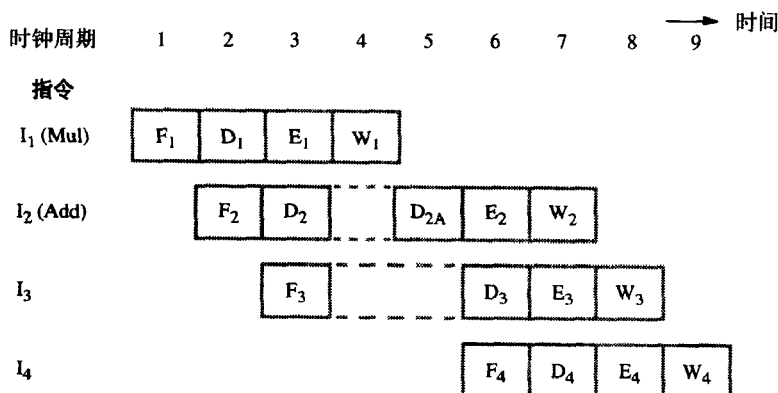


图8-6 由于D<sub>2</sub>和W<sub>1</sub>之间的数据依赖性导致的流水线延迟

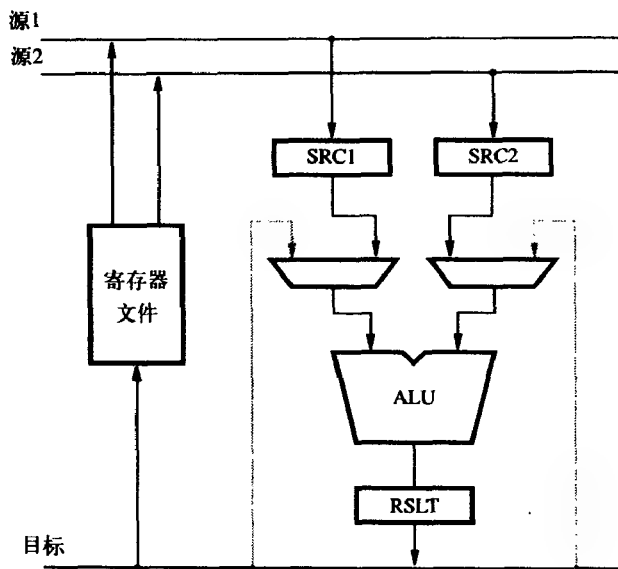
### 8.2.1 操作数传递

产生数据阻塞是因为图8-6中的指令I<sub>2</sub>等待将要写入到寄存器文件的数据。然而，一旦执行阶段完成E<sub>1</sub>步，就可以在ALU的输出端获得这些数据。因此，如果我们安排指令I<sub>1</sub>的结果直接传送到E<sub>2</sub>步使用，则延迟能够被缩短或者可能被消除。

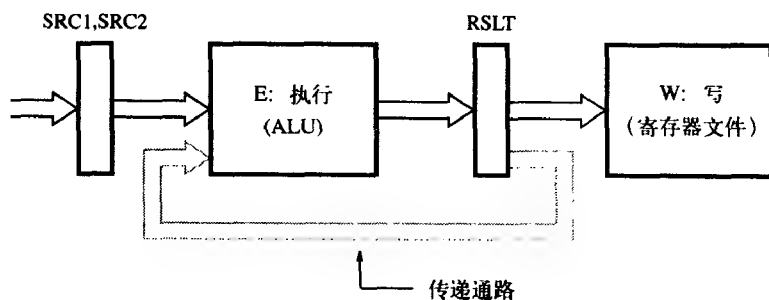
462

图8-7a给出的是包含ALU和寄存器文件的处理器数据通路的一部分。这种方案除增加了SRC1、SRC2和RSLT寄存器以外，与图7-8中的三总线结构相似。这些寄存器构成了流水线操作所需要的中间存储缓冲器，如图8-7b所示。参看图8-2b，寄存器SRC1和SRC2是缓冲器B2的一部分，RSLT是B3的一部分。数据传递机制用浅色线标出。与ALU输入端相连的两个多路复用器允许选择目标总线上的数据来代替SRC1或SRC2寄存器中的数据。

当图8-6中的指令在图8-7中的数据通路上执行的时候，在每一个时钟周期实现的操作如下。指令I<sub>2</sub>译码后检测到数据依赖性，于是决定使用数据传递。寄存器R2中操作数不存在数据依赖性，所以在第3个时钟周期读出并放入寄存器SRC1中。在下一个时钟周期，指令I<sub>1</sub>产生的乘积可以在寄存器RSLT中获得，并且由于是传递连接，这个乘积可在E<sub>2</sub>步使用。因此，I<sub>2</sub>将正常执行而不会被中断。



a) 数据通路



b) 处理器流水线中源寄存器和目标寄存器的位置

图8-7 流水线处理器中的操作数传递

### 8.2.2 用软件方法处理数据阻塞

在图8-6中，我们假设数据依赖性是由硬件在指令译码过程中检测到的。控制硬件延迟到第5个周期才读寄存器R4，这样除非使用操作数传递，否则将引入两周期的延时。另一种可选择的方法是将检测数据依赖性的任务留给软件处理。在这种情况下，编译程序可以插入NOP（无操作）指令在指令I<sub>1</sub>和I<sub>2</sub>之间产生所需要的两个周期的延时，例如：

I<sub>1</sub>: Mul R2, R3, R4

NOP

NOP

I<sub>2</sub>: Add R5, R4, R6

如果检测这种依赖性的任务完全留给软件，那么编译程序必须能够插入NOP指令以获得正确结果。这种可能性说明了编译程序和硬件之间的紧密联系。某些特殊的性能可以由硬件实现，也

可以留给编译程序来实现。将插入NOP指令的任务留给编译程序完成会使硬件更加简化。如果认为需要这种延迟,编译程序可以尝试着对指令重新排序以便在NOP时间段执行其他有用的任务,从而获得较好性能。另一方面,NOP指令的插入使得代码长度增加。而且经常会出现这种情形,一个给定的处理器体系结构具有多种硬件实现方法,不同的实现方法提供不同的特性。为了满足一种实现的需要而插入的NOP指令可能对另一种不同实现方法并不合适,从而会导致性能降低。

### 8.2.3 副作用

在前面例子中遇到的数据依赖性很明显并且很容易检测到,因为涉及到的寄存器在指令 $I_1$ 中作为目标寄存器,而在 $I_2$ 中作为源寄存器。有时候指令还会修改其他寄存器的内容,而不仅仅是修改目标寄存器。使用自动递增或自动递减寻址方式的指令就是这样的一个例子。它除了在目标单元存储新数据,指令还修改了用于访问指令的一个操作数的源寄存器的内容。用于处理涉及目标单元的数据依赖性的所有预防措施,必须也要适用于受自动递增或自动递减操作所影响到的寄存器。除了在指令中明确指明的目标操作数单元以外,其他的单元也受到了影响时,就称该指令有副作用。例如像push和pop这种堆栈指令就产生类似的副作用,因为它们隐含地使用自动递增和自动递减寻址方式。

464

另一种产生副作用的可能性是条件码标志,它被诸如条件转移和带进位的加法等指令所使用。假设在寄存器R1和R2中存有一个双精度整数,我们希望将它与在寄存器R3和R4中的另一个双精度数相加。可能按以下方式实现:

Add R1, R3

AddWithCarry R2, R4

这两条指令通过进位标志存在着隐含的依赖性。进位标志由第一条指令设置并被用于第二条指令中,它执行的操作是

$R4 \leftarrow [R2] + [R4] + \text{进位}$

有副作用的指令会产生大量数据依赖性,导致用来解决数据依赖性的硬件或软件的复杂度大大提高。由于这种原因,设计在流水线硬件上执行的指令应该很少有副作用。理想情况是,只有那些在寄存器或存储单元中的目标单元内容会受到给定指令的影响。应该将诸如设置条件码标志或者更新地址指针内容等副作用降到最低。然而,在第2章给出的自动递增和自动递减寻址方式是很有用的,条件码标志也需要在算术运算中记录产生的进位或溢出等信息。在8.4节中我们会指出这些功能如何由符合流水线结构和优化编译器要求的其他方式提供。

## 8.3 指令阻塞

取指令部件的目标是为执行部件提供一个平稳的指令流,无论何时指令流中断,流水线就会出现延迟,就像在图8-4中说明的高速缓存失效的情形。转移指令也可能会引起流水线的延迟。现在来看看转移指令的影响以及用来减缓转移指令影响的技术。我们先从无条件转移开始。

465

### 8.3.1 无条件转移

图8-8给出的是正在一个2段流水线上执行的指令序列。指令 $I_1$ 到 $I_3$ 存储在地址连续的存储单



元中,  $I_2$  是转移指令。其转移目标是指令  $I_k$ 。在第3个时钟周期取指令  $I_3$ , 同时, 对转移指令进行译码并计算目标地址。在第4个时钟周期, 处理器丢弃已经取出且不正确  $I_3$ , 并取出指令  $I_k$ 。同时通知负责执行 (E) 步骤的硬件部件在那个时钟周期什么都不要做, 从而使流水线延迟一个时钟周期。

由于转移指令而浪费的时间一般叫做转移代价。在图8-8中, 转移代价是一个时钟周期。对于较长的流水线, 转移代价可能会更高。例如, 图8-9a是转移指令在一个4段流水线上执行的效果。假设转移地址在  $E_2$  步计算, 在第5个时钟周期必须将指令  $I_3$  和  $I_4$  丢弃同时取出目标指令  $I_k$ 。因此, 转移代价是两个时钟周期。

为了减少转移代价, 应该在流水线中较早地计算出转移地址。典型地, 在取指令后, 取指令部件指示硬件识别转移指令并且尽可能快地计算出转移目标地址。有了这个额外硬件, 两个任务在  $D_2$  步都能执行。事件的发生顺序如图8-9b所示。在这种情形下, 转移代价仅仅是一个时钟周期。

466

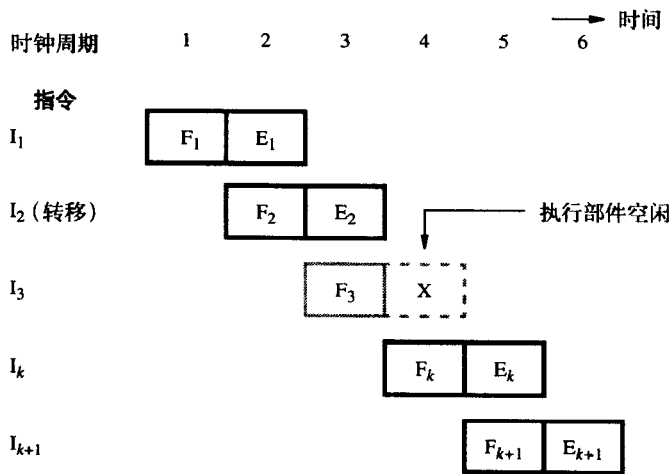


图8-8 由转移指令产生的一个空闲周期

### 指令队列和预取操作

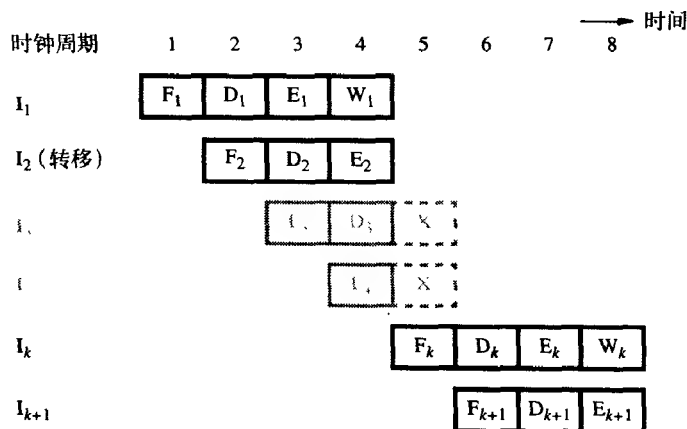
高速缓存失效或转移指令会使流水线延迟一个或多个时钟周期。为了减少这些中断的影响, 许多处理器采用复杂的取指令部件。这种取指令部件能够在指令使用前将指令取出并把它放在一个队列中。通常, 指令队列能够存储多条指令。一个称为调度单元的独立部件从队列前面取出指令并送到执行部件中。这样的硬件结构如图8-10所示。调度单元也执行译码功能。

467

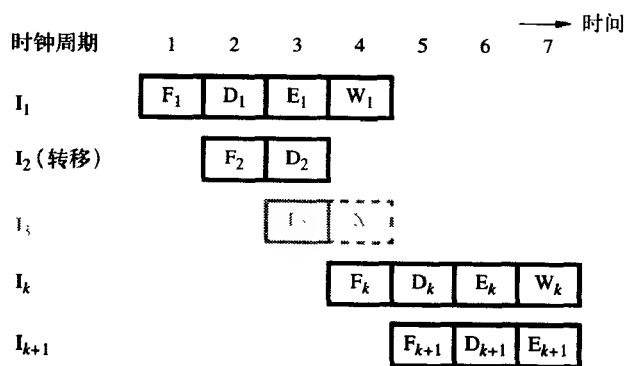
为了有效, 取指令部件必须有充足的译码和处理能力来识别和执行转移指令。它尽量保持指令队列总是满的, 以降低取指令时的偶然延迟所产生的影响。例如, 当由于数据阻塞而带来流水线延迟时, 调度部件不能发送从指令队列中取出的指令。然而, 取指令部件继续取指令并把它加入队列中。相反, 如果由于转移指令或高速缓存失效而使得取指令时存在延迟, 调度部件会继续发送从指令队列中取出的指令。

图8-11说明了队列长度如何改变以及它如何影响不同流水线阶段之间的关系。假设初始队列中包含有一条指令, 每一个取操作在队列中增加一条指令, 每次调度操作使队列长度减1。因此, 在前4个时钟周期中, 队列长度保持不变。(在每一个周期中都有步骤F和步骤D。) 假设指

令 $I_1$ 产生了两周期的延迟。因为队列中有空间，所以取指令部件继续取指令，在第6个时钟周期队列长度增至3。



a) 转移地址在执行阶段计算



b) 转移地址在译码阶段计算

图8-9 转移时序

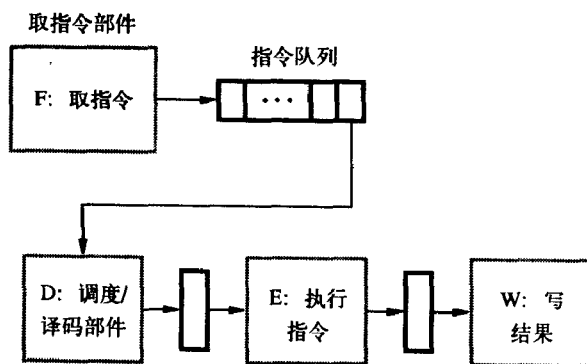


图8-10 在图8-2b的硬件结构中使用一个指令队列

指令 $I_5$ 是一条转移指令。在第7个周期取它的目标指令 $I_k$ 并丢弃指令 $I_6$ 。在第7个周期，由于丢弃了指令 $I_6$ ，转移指令会引起延迟。相反，指令 $I_4$ 从队列调度到译码段。丢弃 $I_6$ 后，在第8个周

期队列的长度降至1。队列长度会保持此值直到遇到另一个延迟。

现在看看在图8-11中指令执行的顺序。指令 $I_1$ 、 $I_2$ 、 $I_3$ 、 $I_4$ 和 $I_k$ 在连续的时钟周期完成执行。因此，转移指令并不增加总的执行时间。这是因为取指令部件在执行其他指令的同时执行了转移指令（通过计算转移地址）。这种技术称为转移重叠。

468

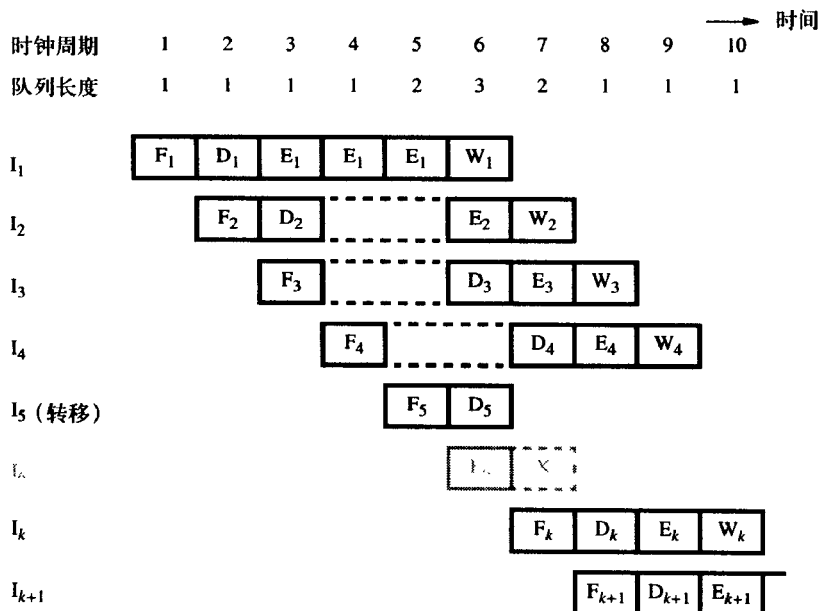


图8-11 存在于一个指令队列中的转移时序；转移目标地址在D段计算

注意转移重叠只有在遇到转移指令且队列中除了转移指令还有至少一条指令的情况下才会发生。如果队列中只有转移指令，那么执行过程就会如图8-9b所示的那样。因此，为了确保为处理过程提供足够的指令，大多数时间应该安排队列是满的。这可以通过提高取指令部件从高速缓存读入指令的速率来实现。在许多处理器中，取指令部件和指令高速缓存之间的连接带宽允许在每一个时钟周期读入多条指令。如果出现转移，取指令部件迅速补充指令队列，那么转移重叠出现的概率就会提高。

增加指令队列在处理高速缓存失效时也是有益的。当发生高速缓存失效时，只要指令队列不空，调度部件就会继续为执行部件传送指令。同时，从主存或二级高速缓存中读入想要的高速缓存块。当取操作重新开始时，就会再填满指令队列。如果队列没有变空，那么高速缓存失效对指令执行的速率就不会有影响。

总之，通过转移重叠过程指令队列减缓了转移指令对性能的影响。它与高速缓存失效引起的延迟效果相似。当取指令部件一次能从指令高速缓存读入多条指令时，该技术的效果会更好。

469

### 8.3.2 条件转移和转移预测

转移条件与上一条指令结果的依赖性使得条件转移指令产生了新增阻塞的可能性。直到相关指令执行结束，才可以做出转移决策。

转移指令会经常出现。实际上，转移指令大约占大多数程序动态指令数的20%（动态数是指令执行的数量，考虑到有些程序指令由于循环而执行很多次的情况）。由于转移代价，这个大

的百分比会降低采用流水线技术所带来的性能提高。幸运的是,可以采用多种方法来处理转移指令以降低它们对指令执行速率的负面影响。

### 延迟转移

在图8-8中,处理器在确定当前指令 $I_2$ 是否是转移指令前就取出了指令 $I_3$ 。当 $I_2$ 执行结束并且出现转移时,处理器必须丢弃 $I_3$ 并取转移目标处的指令。跟在转移指令后面的单元叫做转移延迟槽。根据执行转移指令所用的时间,可能存在多个转移延迟槽。例如,在图8-9a中有两个转移延迟槽,在图8-9b中有一个。在延迟槽中的指令通常在确定转移和转移目标地址被计算出来之前已经被取出,并且至少已经有一部分被执行了。

延迟转移技术可以减小由于条件转移指令而带来的时间代价。这种想法很简单,在延迟槽中的指令通常是可以被提取的。因此,无论转移是否发生,我们希望对它们进行完全的执行。目的是能够在这些槽中放入有用指令。如果在延迟槽中没有放入有用的指令,这些槽必须用NOP指令填满。这种情形与在8.2节讨论的数据依赖性情形完全相同。

考虑在图8-12a中给出的指令序列。寄存器R2当作计数器,它决定寄存器R1的内容左移的次数。对于有一个延迟槽的处理器,指令能够如图8-12b所示重新排序。在转移指令执行的同时读取移位指令。判断转移条件后,处理器根据转移条件是真还是假来读取LOOP或NEXT处的指令。在任何一种情形下,它都执行了移位指令。在循环的后两步,事件的顺序如图8-13所示。在任何时刻,流水线操作没有被中断,并且不存在空闲周期。逻辑上,程序执行就像转移指令放在移位指令之后一样。

也就是说,与转移指令在存储器中的指令顺序相比,要晚一条指令才会发生转移,因此称为“延迟转移”。

延迟转移方法的有效性取决于如图8-12所示的重排指令的频率。从许多程序搜集到的实验数据证明,在85%的情形下复杂的编译技术可以使用一个转移延迟槽。对于拥有两个转移延迟槽的处理器,编译程序尽力找出转移指令之前的两条指令,它将其移入延迟槽而不会产生逻辑错误。找到两条这样指令的几率远远小于找到一条的几率。因此,如果增加一个用来增加转移延迟槽数量的流水线阶段,那么在性能方面的收获或许不能完全体现出来。

### 转移预测

降低与条件转移相关的转移代价的另一种技术是尽力预测出某个特定的转移是否会执行。转移预测的最简单形式是假设不会发生转移,继续以连续地址顺序取指令。直到转移条件被判定,指令沿着预计路径的执行一定是在推测的基础上做出的。推测执行意味着指令在处理器确定它们的正确执行顺序之前已经开始执行。因此,必须注意直到可以确定这些指令的确应该执行后,处理器寄存器或存储器单元才能更新。如果转移决策指明的是另一种情况,那么执行部件中的指令和相关数据必须清除,并取出正确指令来执行。

图8-14所示的是4段流水线的一种不正确的预测转移。该图表示在一条比较指令后面跟着一条Branch>0指令。转移预测在第3个周期发生,同时取指令 $I_3$ 。取指令部件预测转移不会发生,当 $I_3$ 进入译码段时继续取指令 $I_4$ 。在第3个周期末得到比较操作的结果。假设比较操作的结果立

470

LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

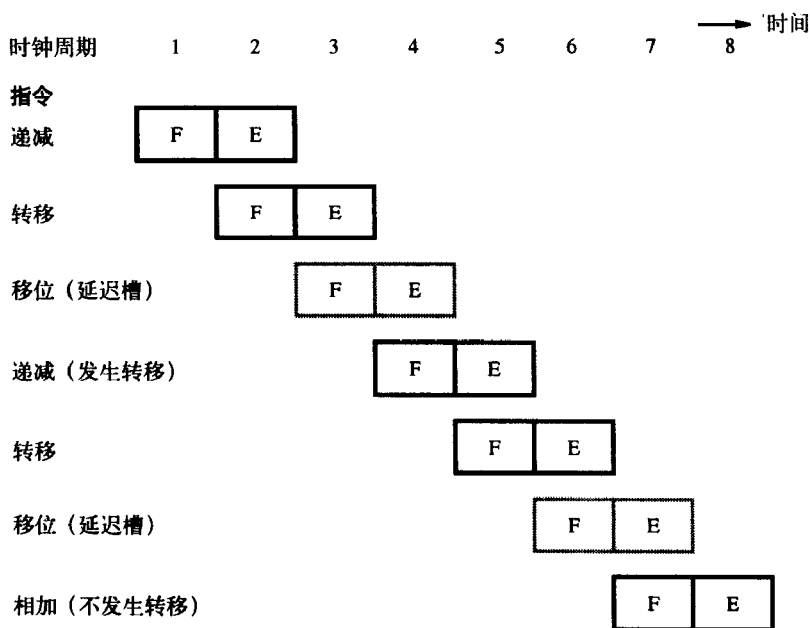
a) 原始循环程序

LOOP	Decrement	R2
	Branch=0	LOOP
	Shift_left	R1
NEXT	Add	R1,R3

b) 重排序的指令

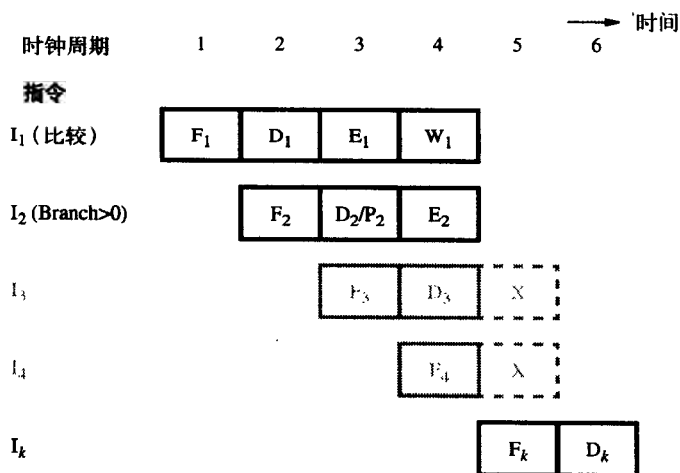
图8-12 用于延迟转移的指令重新排序

即传送给取指部件, 在第4个周期判断转移条件。此处, 取指令部件意识到预测不正确, 并清除执行流水线中的两条指令。在第5个时钟周期从转移目标地址处取出新的指令 $I_k$ 。



471

图8-13 执行时序表明在图8-12b中的最后两个步骤中通过循环填满延迟槽



472

图8-14 当转移决策与预测不相符时的时序

如果转移出口是随机的, 那么有一半的转移会被捕获到。假设不发生转移的简单方法可节省条件转移所浪费时间的50%。然而, 如果根据预期程序的行为, 预测某些转移指令会发生, 而另一些不会发生, 这样就可以获得较好的性能。例如, 循环末尾的转移指令导致在整个循环期间除了最后一步外, 每一步都可使转移回到循环的开始。因此, 假设这个转移会发生, 并且让取指部件开始读取转移目标地址处的指令是有利的。相反, 对于程序循环开始处的转移指令假设转移不会发生是比较有利的。

预测转移的结果, 可以用观察转移目标地址比转移指令的地址低还是高的方法用硬件来实

现。一种比较灵活的方法是由编译程序决定给出的转移指令应该预测为发生还是不发生。在一些处理器的转移指令中,例如SPARC,包括有转移预测位,通过编译程序对它置为0或1来表示所需要的行为是什么。取指令部件检查这个位来预测转移是否会发生。

无论采用哪种策略,对于某个具体指令每次执行时转移预测决策通常是相同的。具有这种特征的任何方法都叫做静态转移预测。另一种方法叫做动态转移预测。在动态预测方法中,预测的决策可能会根据执行过程而改变。

### 动态转移预测

转移预测算法的目标是减少作出错误决策的概率,避免取出那些最后不得不丢弃的指令。在动态转移预测方案中,处理器硬件通过跟踪每次指令执行的结果来判断特定转移发生的可能性。

最简单形式是用给定转移指令出口的执行历史作为这条指令最近执行的结果。处理器假设下一次指令执行时,其结果可能是相同的。因此,算法可以用图8-15a中的2状态机来描述。这两个状态是:

LT: 可能发生转移

LNT: 可能不发生转移

假设算法从状态LNT开始。当转移指令执行时,如果发生转移,状态机移到状态LT。否则,保持在状态LNT上。下一次遇到相同指令时,如果相应的状态机在状态LT,预测为会发生转移,否则预测为不发生转移。

这种简单的方案在循环程序内部会很好地运行,它要求每一条转移指令要有一位历史信息位。一旦进入循环,控制循环的转移指令总是会产生相同的结果直至整个循环的最后一步。在最后一步,转移预测会转向一个错误处,并且转移历史状态机也会改变到相反的状态上。假设存在多遍循环,则下一次进入同样的循环时,状态机会产生错误预测。

对执行历史保留更多信息可以获得较好的性能。采用4状态算法,每一条转移指令需要两个历史信息位,如图8-15b所示。这4个状态是:

ST: 极有可能发生

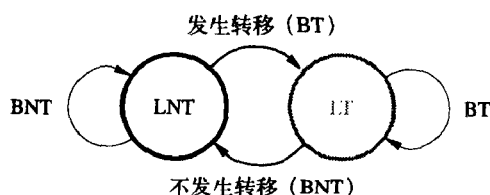
LT: 有可能发生

LNT: 不可能发生

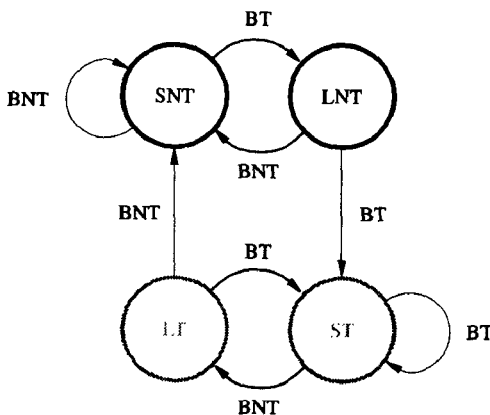
SNT: 极不可能发生

假设算法的初始状态设置为LNT。执行转移指令后,如果确实发生转移,那么状态变化到ST;否则,变化到SNT。随着程序的执行,再一次遇到同样的指令时,转移预测算法的状态继续如图所示发生变化。当遇到转移指令时,如果状态是LT或ST,那么取指部件预测会发生转移,并开始取转移目标地址处的指令。否则,继续取连续地址处的指令。

了解转移预测算法的具体行为是有意义的。当处于状态SNT时,取指部件预测不会发生转移。如果实际上发生了转移,即如果预测不正确,那么状态变化到LNT。这意味着下一次遇到



a) 2状态算法



b) 4状态算法

图8-15 转移预测算法的状态机表示

相同转移指令时，取指部件将会预测不发生转移。只有在一行中两次预测不正确时状态才会变化到ST。从那以后，预测为会发生转移。

我们再考虑一下当执行循环程序时所发生的情况。假设转移指令位于循环末尾并且处理器设置算法的初始状态为LNT。在循环的第一遍，预测将是错误的（不发生转移），从而状态会变化到ST。在后续的所有循环中，除了最后一遍，预测都将是正确的。那时，状态会变化到LT。当再一次进入循环时，预测会是正确的（发生转移）。

现在进行最后一次修改以更正首次进入循环时的错误转移预测。在这种情形下预测错误的原因是转移预测算法的初始状态。由于缺少转移指令属性的附加信息，我们假设处理器设置的初始状态为LNT。所需要的设置初始状态的正确信息可以由前面讨论的任何一种静态预测方案提供。处理器可以通过比较地址或者通过检查指令中的预测位，来设置算法的初始状态为LNT或LT。在循环末发生转移时，编译程序会指出转移应被预测为会发生转移，将初始状态设置为LT。有了这个修改，除了整个循环的最后一步外，转移预测通常会是正确的。而后一种错误预测的情况是不可避免的。

处理器可以采用多种方法来保留用于动态转移预测算法的状态信息。它们可以被记录在一张查询表中，由转移指令地址的低位部分存取。在这种情形下，可能会有两条转移指令共享同一个表的入口。这可能导致转移预测错误，但是它不会导致执行错误。错误预测只会引入执行时间的小延迟。一种替换的方法是在指令高速缓存中存储与转移指令相关的历史信息位作为标签。在8.7节我们会看到在SPARC处理器中这条信息是如何处理的。

## 8.4 对指令集的影响

我们已经了解到有一些指令比较适合于流水线执行，而另一些则不适合。比如，指令副作用可能会产生不希望的数据依赖性。在本节来了解流水线执行和机器指令特性之间的关系。我们讨论机器指令的两个关键方面——寻址方式和条件码标志。

### 8.4.1 寻址方式

寻址方式应该提供简单有效的存取各种数据结构的方式。比较常用的寻址方式包括变址寻址、间接寻址、自动递增寻址和自动递减寻址。许多处理器提供这些方式的各种组合以增加它们指令集的灵活性。复合寻址方式经常会遇到，比如那些涉及双变址的方式。

选择在流水线处理器中实现的寻址方式时，我们必须考虑每一种寻址方式对流水线指令流的影响。这方面要考虑的两个重要内容是：自动递增寻址和自动递减寻址方式的副作用以及复合寻址方式引起的流水线拖延的程度；另一个重要因素是编译程序是否能利用给定的寻址方式。

为了比较各种方法，我们假定一种用于访问存储器中操作数的简单模型。取指令 Load X (R1), R2 用5个周期完成它的执行，如图8-5所示。然而，指令

Load (R1), R2

由于不需要计算地址，从而可以构成适应于4阶段流水线的方式。对存储器的访问可在E段发生。比较复杂的寻址方式可能需要多次访问存储器来获得所需要的操作数。例如，指令

Load (X (R1)), R2

可能会如图8-16a所示那样执行，这里假设变址偏移量X在指令字中给出。在第3个周期完成计算

地址后,处理器需要两次访问存储器——第一次在第4个时钟周期读单元 $X+[R1]$ ,接着在第5个周期读单元 $[X+[R1]]$ 。如果 $R2$ 是下一条指令中的源操作数,那么下条指令将会延迟3个周期,用操作数传递可缩短到2个周期,如图所示。

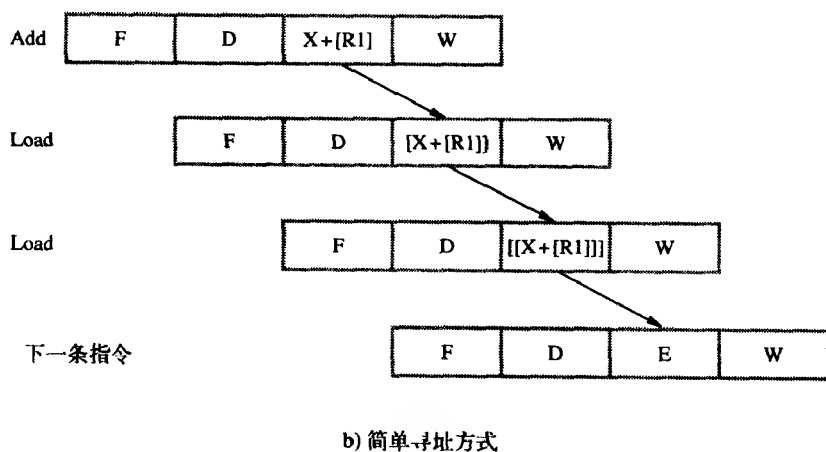
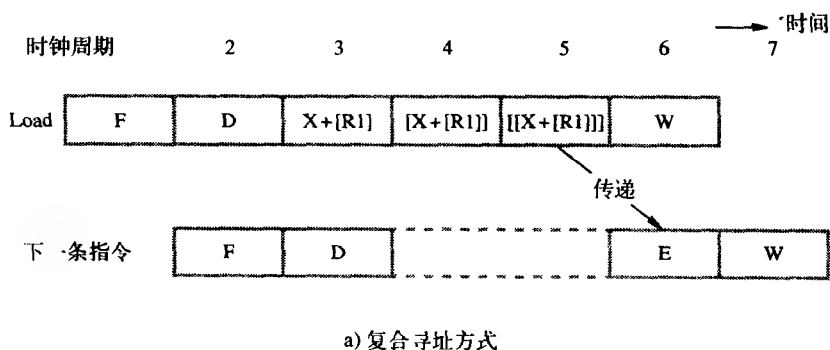


图8-16 采用复合和简单寻址方式的等效操作

如果只使用简单寻址方式来实现同样的Load操作,就需要多条指令。例如,在允许三操作数地址的计算机中,我们可用

Add #X, R1, R2

Load (R2), R2

Load (R2), R2

Add指令执行操作 $R2 \leftarrow X+[R1]$ 。这两条取指令先后从存储器中取地址和操作数。这个指令序列正好与最初的单条取指令使用的时钟周期数相同,如图8-16b所示。

这个例子表明,在流水线处理器中,涉及多次访问存储器的复合寻址方式不一定会提高执行速度。此类方式的主要优点是减少了执行给定任务所需要的指令数量,从而减少了在主存储器中所需要的程序空间。主要缺点是执行时间长,造成流水线拖延,从而降低流水线的效率。复合寻址方式需要用较复杂的硬件进行译码和执行。而且,复合寻址方式也不利于编译程序的工作。

现代处理器的指令集通常设计成最大限度利用流水线优势的硬件。因为复合寻址方式不适



应于流水线的执行, 所以应该避免。用于现代处理器的寻址方式一般具有下列特征:

- 存取操作数要求访问存储器最多一次。
- 只有取和存指令可以访问内存操作数。
- 采用的寻址方式没有副作用。

具有这些特征的三种基本寻址方式是寄存器寻址、寄存器间接寻址和变址寻址。前两种不需要地址计算。变址寻址方式的地址可在一个周期中计算出来, 无论变址值是在指令中给出还是在寄存器中给出。存储器的访问在随后的一个周期中进行。这三种方式不能有任何的副作用和任何一种可能的异常。有一些体系结构, 例如ARM, 允许变址方式下计算出的地址写回到变址寄存器中。这是在上面的指导思想下不允许有的副作用。而且还要注意可能会用到相对寻址, 这是变址寻址中的一种特殊情况。在这种方式下程序计数器当作变址寄存器。

这里列出的3个特征第一次作为RISC处理器概念中的一部分被提出。SPARC处理器体系结构遵循这些特征, 这些将在8.7节中介绍。

#### 8.4.2 条件码

就像在第3章中描述的那样, 在许多处理器中条件码标志存储在处理器的状态寄存器中。它们可以被许多指令设置或者清除, 以便由后续的条件转移指令来测试从而改变程序执行的流程。流水线处理器的优化编译器试着对指令重新排序, 以避免当连续指令之间出现转移或数据依赖性时使流水线出现延迟。在实现这个功能时, 编译器必须保证重新排序的指令序列不会导致计算结果的改变。由条件码标志产生的依赖性降低了编译器对指令重新排序的灵活性。

478

考虑图8-17a中的指令序列, 假设Compare指令和Branch=0指令的执行按图8-14那样进行。因为转移决策必须等待Compare指令的结果, 所以转移在步骤E<sub>2</sub>而不是D<sub>2</sub>发生。可以通过互换Add指令 and Compare指令来减小Branch指令的执行时间, 如图8-17b所示。这会使转移指令相对于Compare指令延迟1个周期。从而, Branch指令正在译码时就能得到Compare指令的结果, 并且作出正确的转移决策, 并不需要转移预测。但是, 只有在Add指令不影响条件码的条件下, 才能互换Add指令和Compare指令。

Add	R1,R2
Compare	R3,R4
Branch=0	...

a) 一个程序片段

Compare	R3,R4
Add	R1,R2
Branch=0	...

b) 重排序的指令

图8-17 指令重排序

通过以上的观察, 我们就处理条件码的方法得出两个重要的结论。首先, 要提供重新排序指令的灵活性, 条件码标志应该尽可能少受指令的影响。其次, 编译器应能够指出程序中的条件码受到哪条指令的影响, 以及不受哪条指令的影响。带有流水线思想设计的指令集通常提供比较理想的灵活性。图8-17b是重新排序的指令, 它假设条件码标志只有在明确指出是作为指令OP码中的一部分时才受影响。SPARC和ARM体系结构提供这种灵活性。

#### 8.5 数据通路和控制

我们在第7章中介绍了处理器内部数据通路的组成结构。考虑图7-8给出的三总线结构。为了使它适用于流水线执行, 可以将其修改成如图8-18所示的形式, 其支持4段流水线。根据使用的寻址方式和实现细节的不同, 数据高速缓存中的操作可能会在E段或之后的段中发生。可以注意到它对图7-8所做的几个重要改变:

479

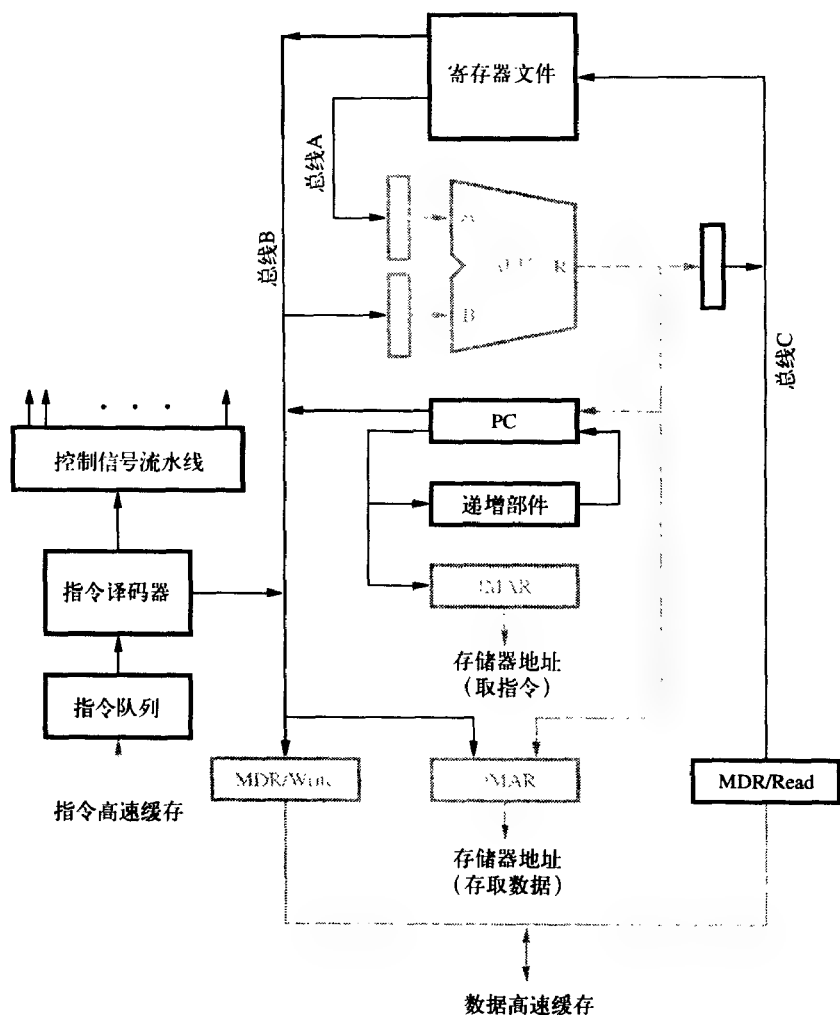


图8-18 为流水线执行而修改的数据通路，在ALU的输入和输出端有中间存储缓冲器

1. 具有单独的指令和数据高速缓存，它们分别通过单独的地址和数据线与处理器相连。这就要求有两种MAR寄存器，IMAR用来访问指令高速缓存，DMAR用来访问数据高速缓存。

2. PC直接连接到IMAR，以便当独立的ALU操作发生时PC的内容能够传送到IMAR中。

3. DMAR中的数据地址可直接从寄存器文件或者ALU中获得，以支持寄存器间接和变址寻址方式。

4. 对于读写操作采用独立的MDR寄存器。在存取操作期间，数据能够直接在这些寄存器和寄存器文件之间传送而不需经过ALU。

5. 在ALU的输入和输出端引入缓冲寄存器。图8-7中的寄存器是SRC1、SRC2和RSLT。在图8-18中不包括转发连接。如果需要可以加上。

6. 指令寄存器被指令队列取代，指令队列从指令高速缓存中取指令。

7. 指令译码器的输出连接到控制信号流水线。缓冲控制信号以及把控制信号随着指令从一个段传送到下一段的要求在8.1节讨论过。在图8-2a中流水线保存着缓冲器B2和B3中的控制信号。

下列操作能够在图8-18中的处理器中独立执行:

- 从指令缓存中读指令。
- 使PC增值。
- 指令译码。
- 从数据高速缓存中读或向数据高速缓存中写。
- 从寄存器文件读两个以上寄存器的内容。
- 向寄存器文件中写入一个寄存器。
- 执行ALU操作。

由于这些操作没有使用任何共享资源, 所以它们能以任何一种组合方式同时执行。该结构提供了实现图8-2中4段流水线所需要的灵活性。例如, 假设 $I_1$ 、 $I_2$ 、 $I_3$ 和 $I_4$ 是4条指令序列。如图8-2a所示, 下列活动在4个时钟周期中都会发生:

- 把指令 $I_1$ 的结果写入到寄存器文件中。
- 从寄存器文件中读指令 $I_2$ 的操作数。
- 对指令 $I_3$ 进行译码。
- 取指令 $I_4$ 并递增PC的值。

## 8.6 超标量操作

流水线使指令的并发执行成为可能。在流水线中同时存在多条指令, 不过它们位于执行的不同阶段。当一条指令在执行ALU操作时, 另一条指令正在译码而又有一条正从存储器中取出。  
481 指令按照严格的程序顺序进入流水线。没有阻塞时, 在每一个时钟周期有一条指令进入流水线, 并有一条指令执行完成。这意味着流水线处理器的最大吞吐量是每个时钟周期一条指令。

一种更先进的方法是使处理器配备有大量的处理部件, 以便在每一处理阶段并行处理多条指令。有了这种方案, 多条指令在同一个时钟周期开始执行, 这种处理器称为使用了多发操作。这种处理器指令执行的吞吐量能够达到每周期执行多条指令, 它们通常被称为超标量处理器。许多现代高性能处理器就是采用这种方式。

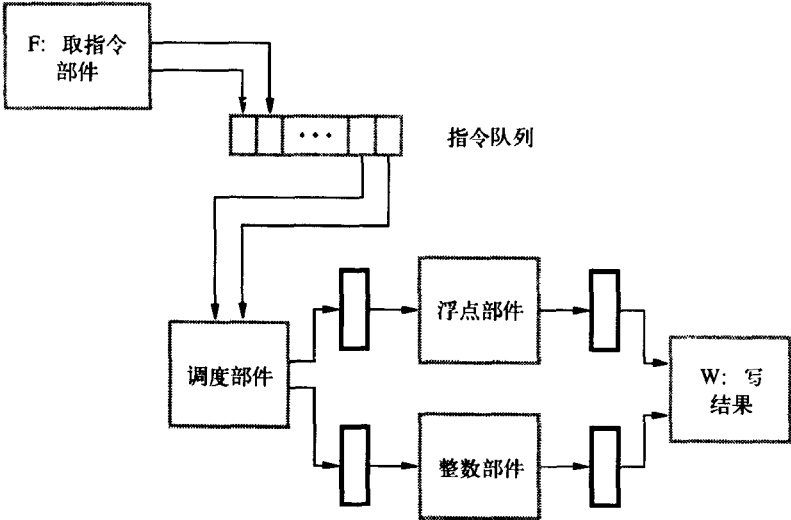
在8.3节介绍了指令队列的思想。我们指出为了保持指令队列是满的, 处理器应该能够一次从高速缓存中取出多条指令。对于超标量操作, 这种安排是必要的。多发操作要求与高速缓存之间要有较宽的通路, 还要有大量的执行部件。同时, 还要为整数指令和浮点指令提供单独的执行部件。

图8-19给出了包含有两个执行部件的处理器例子, 一个用于整数操作, 一个用于浮点操作。取指令部件能够一次读取两条指令并把它们存储在指令队列中。在每一个时钟周期, 调度部件从队列头部取出两条指令进行译码。如果一条是整数指令, 一条是浮点指令, 就不会存在阻塞问题, 那么这两条指令就可以在同一时钟周期内进行调度。

在超标量计算机中, 各种阻塞对性能的负面影响变得更加明显。编译器能够通过合理的选择和指令排序来避免许多阻塞。例如, 对于图8-19中的处理器, 编译器应该尽量交叉浮点和整数指令。这样能够使调度部件保持整数和浮点部件在大部分时间都处于工作状态。总之, 如果编译器能够最大限度地利用可获得的硬部件来安排程序指令, 就可以获得性能上的提高。

流水线时序如图8-20所示。阴影部分表示在浮点部件中的操作。浮点部件利用3个时钟周期来完成在 $I_1$ 中规定的浮点操作。整数部件在一个时钟周期内完成 $I_2$ 的执行。还假定浮点部件内部

是由3段流水线构成的。因此，在每一个时钟周期浮点部件还能够接受一条新指令。从而，指令 $I_3$ 和 $I_4$ 在第3个周期进入调度部件，两者都在第4个周期调度。整数部件能够接受一条新指令，因为指令 $I_2$ 已进行到写阶段。指令 $I_1$ 仍在执行阶段，但是它已移到浮点部件内部流水线的第二阶段中。因此，指令 $I_3$ 能够进入第一阶段。假设没有遇到阻塞，指令按照图示方式完成执行。



482

图8-19 含有两个执行部件的处理器

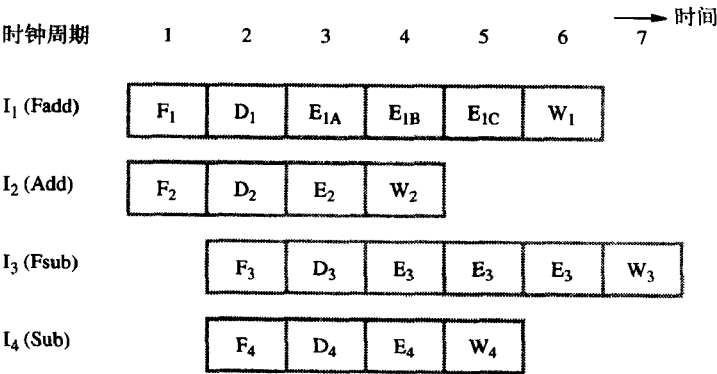


图8-20 在图8-19的处理器中的指令执行流的例子，假设没有遇到阻塞

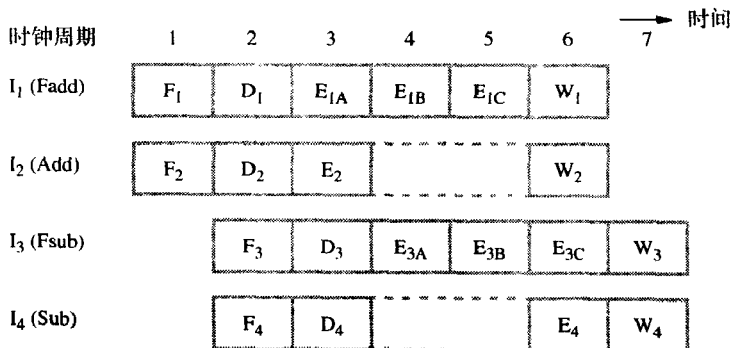
8.6.1 无序执行

在图8-20中，按照指令在程序中出现的顺序进行调度。然而，指令的执行是无序的。这样会带来一些问题吗？我们已经讨论过指令之间的依赖性所产生的问题。例如，如果指令 $I_2$ 依赖 $I_1$ 的结果，那么 $I_2$ 的执行就会被延迟。只要这种依赖性能够被正确处理，就不必延迟指令的执行。然而，在考虑指令引起异常的概率时又出现了新的困难。引起异常的原因可能是取操作数期间的总线错误，或者是非法操作导致的，比如被0除。在第4个周期 $I_2$ 的结果写回到寄存器文件中，如果是指令 $I_1$ 引起的异常，那么程序的执行就会出现不一致的状态。程序计数器指向的指令是异常发生处的指令。但是，一个或多个成功的指令已经执行完成。如果这种状况允许，就说该

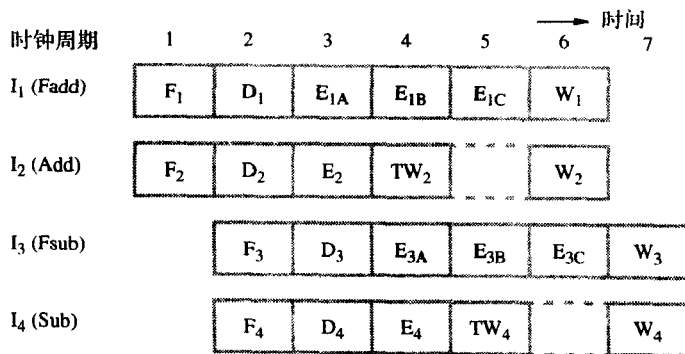
483

处理器有不精确异常。

为了保证异常发生时状态的一致性,指令的执行结果必须严格按程序顺序写入到目标位置中。这意味着必须延迟图8-20的W<sub>2</sub>段直到第6个周期,才能接受指令I<sub>4</sub>,如图8-21a所示。如果在指令期间发生异常,那么丢弃可能已经部分执行的所有后续指令,这被称作精确异常。



a) 延迟写



b) 使用临时寄存器

图8-21 指令按程序顺序完成

在外部中断的情况下很容易提供精确异常。当收到外部中断时,调度部件停止从指令队列中读取新指令,丢弃队列中剩余的指令。所有未执行完的指令继续执行。在这一点,处理器和所有寄存器都处于一致的状态,可以对中断进行了。

### 8.6.2 执行完成

我们希望使用无序执行,它可以使执行部件尽可能自由地执行其他指令。同时,指令必须按程序顺序完成以允许精确异常。如果允许图8-20所示的那种情况执行,这些表面上的冲突要求就很容易解决,但是结果要写到临时寄存器。临时寄存器的内容按正确的程序顺序传送到永久寄存器中。这种方法如图8-21b所示。TW步向临时寄存器写入。W步是最后一步,在这一步中临时寄存器的内容传送到合适的永久寄存器中。这一步通常叫做提交步,因为从那点后指令的影响不能再被恢复。如果是由某条指令造成异常,那么任何已经执行的后续指令的结果仍会

保存在临时寄存器中,并且可以安全地丢弃。

临时寄存器承担永久寄存器的作用,保存永久寄存器的数据并且指定为同样的名称。例如,如果 $I_2$ 的目标寄存器是R5,那么在第6个和第7个时钟周期,在 $TW_2$ 步用到的临时寄存器就作为R5来对待。在这期间,临时寄存器的内容可传送给任何指向R5的后续指令。由于这个特性,这项技术被称为寄存器重命名。注意临时寄存器只用于按程序顺序跟在 $I_2$ 之后的指令。如果 $I_2$ 前一条指令需要在第6个或第7个周期读R5,那么它实际上真正读的是寄存器R5,这个寄存器中仍然保存着未被指令 $I_2$ 修改的数据。

当允许无序执行时,需要一个专门的控制部件来保证指令按照正确的顺序提交。这个部件称为提交部件。它使用一个称为重新排序缓冲器的队列来决定下一次应该提交哪条(哪些)指令。随着指令不断的调度执行,指令严格地按照程序顺序进入队列。当一条指令到达队首并且该指令的执行已经完成时,相应的结果从临时寄存器传送到永久寄存器中,指令从队列中清除。释放所有分配给该指令的资源,包括临时寄存器。这时就说该指令已经被释放了。因为只有指令位于队首时才会被释放,在它之前调度的所有指令也一定都被释放了。因此,指令可以按照无序方式结束执行,但必须按照程序的顺序来释放。

485

### 8.6.3 调度操作

我们现在讨论调度操作。当做出调度决策时,调度部件必须确保已经具备了指令执行所需要的所有资源。例如,由于指令的执行结果可能需要写到临时寄存器,所以所需要的寄存器必须是空闲的,并且保留出来用作那条指令进行调度操作的一部分。在重新排序缓冲器中也必须为该指令留出位置。当分配完所需要的所有资源包括相应的执行部件时,便可以开始调度这条指令了。

指令可以按照无序方式进行调度吗?例如,在图8-20b中指令 $I_2$ 由于源操作数的高速缓存失效而延迟,整数部件在第4个周期正在工作而无法调度 $I_4$ ,那么这时可以对 $I_5$ 进行调度吗?理论上这是可以的,条件是在重新排序缓冲器中为指令 $I_4$ 预留出空间,以确保所有指令按正确次序释放。无序调度指令需要相当小心。如果在 $I_4$ 等待某些资源时对 $I_5$ 进行调度,必须确保不会发生死锁。

死锁是当两个部件A和B使用同一共享资源时出现的一种状态,假设B部件要等待A部件完成后才能完成它的任务。同时,A部件所需要的资源已经分配给了B部件。如果这种情况发生,那么两个部件都不能完成任务。A部件等待它所需要的资源,而该资源正被B部件占有。同时,B部件在释放该资源之前等待A部件完成。

如果指令的调度是无序的,以下的方式可能会出现死锁。假设处理器在调度 $I_5$ 时只有一个临时寄存器,该寄存器留给 $I_5$ 使用。由于指令 $I_4$ 正在等待临时寄存器而无法调度,而临时寄存器直到指令 $I_5$ 释放后才会空闲。但是指令 $I_5$ 不能在 $I_4$ 之前释放,所以产生死锁。

为了预防死锁,调度必须考虑许多因素。因此,无序调度指令可能会大大增加调度部件的复杂度。这也意味着可能需要更多的时间做出调度决策。由于这些原因,大多数处理器只使用有序调度。因此,指令的调度和指令释放都是按照程序中的顺序进行的。在这两事件之间,几条指令的执行可以按它们自己的速度进行,只有指令间的内部依赖性会影响到它们。

在下一节,我们将以UltraSPARC II作为范例来介绍商业上成功的超标量流水线处理器。在本章出现的各种问题在该处理器中都得到了解决,并且它的选择方案很有指导意义。

## 8.7 UltraSPARC II 实例

处理器设计在近几年有很大的提高。以纯RISC或纯CISC作为处理器分类依据已不再适合,因为现代高性能处理器中都包含这两种设计风格。

早期的RISC处理器说明有很多特性可以对高性能产生影响。下列两项观察结果是非常重要的:

- 流水线, 它使处理器能够同时执行数条指令, 只要流水线不经常出现延迟, 性能就可以得到很大的提高。
- 硬件和编译器设计之间的紧密配合, 使编译器通过减少引起流水线拖延的事件最大限度地利用流水线结构。

正是这些因素而不是简单的精简指令集促成了RISC处理器的成功。在这方面尤其重要的是硬件设计之间的紧密合作, 尤其是流水线结构和编译器设计之间的紧密协调。当今的处理器都具有很高的性能, 这主要归因于编译技术的发展, 编译技术继而又引发了新的硬件特性, 这些新的硬件特性在过去几年里可能是毫无用处的。

SPARC体系结构是SUN工作站使用的处理器基础, 它是一个非常优秀的例子。SUN所实现的一款SPARC体系结构的处理器叫做UltraSPARC II, 也正是我们将要讨论的处理器。选择这款处理器而没有选择第3章中介绍的处理器, 是因为它可以很好地说明超标量操作以及在本章讨论过的大多数流水线设计选择和权衡。我们先从SPARC体系结构的简单介绍开始。要了解完整的描述, 读者可查阅SPARC体系结构手册<sup>[1]</sup>。

### 8.7.1 SPARC体系结构

SPARC即可扩缩处理器体系结构 (Scalable Processor ARChitecture)。它是处理器指令集体系结构的具体规格, 也就是说, 它是处理器指令集和寄存器组织的规格, 无论这些指令集和硬件组织是如何用硬件实现的。而且, SPARC是“开放的体系结构”, 这意味着除了SUN公司, 其他计算机公司可以开发它们自己的硬件来实现相同的指令集。

SPARC体系结构在1987年首次公布, 它源自于80年代早期在加州大学伯克利分校的一个项目中的开发思想。在这个项目中产生了精简指令集计算机以及相应的缩写“RISC”。SUN公司和其他几个处理器芯片制造商已经设计并生产了基于这种体系结构的许多处理器, 涵盖了各种性能级别。SPARC体系结构规格由国际协会控制, 它每隔几年引入新的增强版本。最新版本是SPARC-V9。

SPARC体系结构的指令集具有显著的RISC特性。该体系结构规格描述了一个数据和存储器地址都是64位长的处理器。指令长度是相同的, 都是32位长, 并提供整数指令和浮点指令。

有两个寄存器文件, 一个用于整型数据, 另一个用于浮点数据。整数寄存器长度为64位。它们的编号根据实现的不同, 可以使用从64到528的数字。SPARC使用称为寄存器窗口的技术。在任何给定的时间里, 应用程序只看到32个寄存器, 即R0到R31。在这些寄存器中, 前8个是总可以被访问的全局寄存器, 其余24个是局部寄存器。

浮点寄存器只有32位长, 根据第6章描述的IEEE标准, 这是单精度浮点数的长度。指令集包括用于双精度和四倍精度操作的浮点指令。用编号连续的两个浮点寄存器存储双精度操作数, 用编号连续的4个浮点寄存器存储四倍精度操作数。总共有64个寄存器, F0到F63。单精度操作数可以存储在F0到F31, 双精度操作数在F0, F2, F4, ..., F62, 四倍精度操作数在F0, F4, F8, ..., F60中。

### 存、取指令

只允许存或取指令对存储器进行访问,在存储器中操作数可以是8位的字节、16位的半字或32位的字。存、取指令也处理64位数。这个64位数被分成两类:扩展字或双字。LDX (Load extended) 指令将称为扩展字的64位数装入到处理器的一个整数寄存器中。双字由两个32位字构成。利用一条LDD (Load double) 指令将这两个字装入到编号连续的处理器寄存器中。它们被放到每个寄存器的低32位,高位用0补充。其中,第一个寄存器是指令中指定的寄存器,它的号码必须是偶数。处理双字的存取指令在处理存储器和浮点寄存器之间传送多精度浮点操作数时是很有用的。

存、取指令使用的寻址方式是两种变址寻址方式中的一种,如下:

1. 有效地址是两个寄存器内容之和:

$$EA = [Radr1] + [Radr2]$$

2. 有效地址是一个寄存器内容和指令中的立即数相加的和:

$$EA = [Radr1] + Immediate$$

488

对于大多数指令,立即数是一个13位有符号数。它是完成了符号扩展到64位后与Radr1中的内容相加的。

使用第一种寻址方式的取指令写作

Load [Radr1 + Radr2], Rdst

它生成有效地址[Radr1] + [Radr2],并把该单元的内容装入Rdst寄存器中。对于立即偏移量的情况,Radr2由立即数来替代,即

Load [Radr1 + Imm], Rdst

存储指令使用类似语法,第一个操作数表示源寄存器,数据从这里存储到存储器,如下:

Store Rsrc, [Radr1 + Radr2]

Store Rsrc, [Radr1 + Imm]

在SPARC指令推荐的语法中,寄存器由%后跟着寄存器号表示。%r2或%2都表示寄存器号2。然而,为了便于阅读并且与前几章保持一致,我们用R0, R1等来表示整数寄存器,用F0, F1, ...来表示浮点寄存器。

作为实例,考虑取无符号字节指令

LDUB [R2 + R3], R4

这条指令从存储器单元[R2]+[R3]处取出一个字节放到寄存器R4的低8位,高56位用0填充。取有符号字指令

LDSW [R2 + 2500], R4

从单元[R2] + 2500读一个32位字,符号扩展到64位,然后把它存储到寄存器R4中。

### 算术和逻辑指令

该处理器还提供一般的算术和逻辑指令集。在表8-1中给出了一些例子。我们在8.4.2节指出,一条指令只有后续的条件转移指令对条件码进行测试时才设置条件码标志。这最大限度地增加了编译器在重排序指令方面的灵活性。SPARC指令集就是秉承这一特征进行设计的。其中,有



两种算术和逻辑指令，一种设置条件码标志，另一种不设置。操作码的后缀cc用于指示应设置的标志位。例如，指令ADD、SUB、SMUL（带符号乘）、OR和XOR不影响标志位，而ADDcc和SUBcc则影响标志位。

寄存器R0的值总是0。当它用作目标操作数时，指令的结果将丢弃。例如，指令

SUBcc R2, R3, R0

将R2的内容减去R3的内容，并设置条件码标志，丢弃减操作的结果。实质上，这是一条比较指令，它的替代语法是

489

CMP R2, R3

在SPARC术语中，CMP叫做合成指令。它不是真正的可由硬件识别的指令，只是为了程序员方便而提供。编译器将采用SUBcc指令代替CMP指令。

表8-1 SPARC指令实例

指 令		描 述
ADD	R5, R6, R7	整数相加: $R7 \leftarrow [R5] + [R6]$
ADDcc	R2, R3, R5	$R5 \leftarrow [R2] + [R3]$ , 设置条件码标志
SUB	R5, Imm, R7	整数相减: $R7 \leftarrow [R5] - \text{Imm}$ (扩展符号)
AND	R3, Imm, R5	按位与: $R5 \leftarrow [R3] \text{AND Imm}$ (扩展符号)
XOR	R3, R4, R5	按位异或: $R5 \leftarrow [R3] \text{XOR}[R4]$
FADDq	F4, F12, F16	浮点数相加, 四倍精度: $F12 \leftarrow [F4] + [F12]$
FSUBs	F2, F5, F7	浮点数相减, 单精度: $F7 \leftarrow [F2] - [F5]$
FDIVs	F5, F10, F18	浮点数相除, 单精度: $F18 \leftarrow [F5] / [F10]$
LDSW	R3, R5, R7	$[R3] + [R5]$ 处32位字符符号扩展为64位
LDX	R3, R5, R7	$[R3] + [R5]$ 处64位扩展字
LDUB	R4, Imm, R5	从存储器单元 $[R4] + \text{Imm}$ 取无符号字节, 该字节被装入寄存器R5的低8位, 而高位被0填充
STW	R3, R6, R12	从寄存器R3存储字到存储器位置 $[R6] + [R12]$
LDF	R5, R6, F3	在地址 $[R5] + [R6]$ 处取32位字, 并装入浮点寄存器F3
LDDF	R5, R6, F8	在 $[R5] + [R6]$ 处取双字, 装入浮点寄存器F8和F9
STF	F14, R6, Imm	从浮点寄存器F14存储字到存储器单元 $[R6] + \text{Imm}$
BLE	icc, Label	检测icc标志, 如果小于或等于0, 转移到Label
BZ, pn	xcc, Label	检测xcc标志, 如果等于0, 转移到Label, 预测转移不发生
BGT, a, pt	icc, Label	检测32位整数标志位, 如果大于0, 转移到Label, 设置取消位, 预测转移发生
FBNE, pn	Label	检测浮点状态标志, 如果不相等则转移; 取消位被置为0, 预测转移不发生

490

条件码寄存器CCR中包含两组条件码标志icc和xcc，分别用于整数和扩展条件码。每组条件码都是由4个标志位N、Z、V和C构成。设置条件码标志的指令，例如ADDcc，可以设置icc和xcc位；xcc标志的设置是基于指令的64位结果，icc标志的设置仅仅参考低32位。用于浮点操作的条件码存储在一个64位寄存器中，该寄存器叫做浮点状态寄存器（FSR）。

### 转移指令

转移处理中采用的方式是决定性能的一个重要因素。SPARC指令集中的转移指令具有若干个特征，这些特征用来提高流水线处理器的性能，并有助于编译器优化输出代码。

SPARC处理器使用带有延迟槽（参见8.3.2节）的延迟转移。转移指令包括转移预测位，编译器利用这些预测位来通知硬件所期望的转移行为。转移指令还包含一个无效位，用来提高处

理延迟槽中指令的灵活性。这条指令总是可以执行的，但是它不输出结果直到得到转移决策后才提交。如果需要进行转移，将延迟槽中的指令执行完并对结果进行处理。如果无须转移，若取消位等于1则该指令将被取消。否则，完成指令的执行。

无论是否执行转移，编译器可以在延迟槽中放入一条所需要的指令。它可能是一条逻辑上在转移指令之前，但却可以移到延迟槽中的指令。在这种情况下无效位应该设置为0。否则，只要需要转移，延迟槽中应该填入一条要执行的指令，此时无效位应该设置为1。

条件转移指令可以测试icc、xcc或FSR标志。例如，指令

```
BGT, a, pt icc, Label
```

如果设置icc标志位的前一条指令产生大于零的结果，将会转移到单元Label处。指令将无效位和转移预测位都设置为1。指令

```
FBGT, a, pt Label
```

除了要测试FSR标志以外，与前面的指令完全相同。如果未指定pt (predicted taken, 采用预测) 或pn (predicted not taken, 不采用预测)，那么编译器默认认为是pt。

491

图8-22给出一个例子来说明转移指令中的预测和取消设施，这是一个将列表中的n个64位整数相加的循环程序。假设表项数目作为一个64位整数存在地址LIST处，该数后面是连续的64位的被加数。还假设列表中至少存在一个数据项，并且程序中的地址LIST已被装入寄存器R3中。

图8-22a给出的是为了在非流水线处理器上执行而写的循环程序。为了在SPARC处理器上执行，应该首先重新组织指令以便有效地利用转移延迟槽。注意每遍循环都要执行LOOPSTART后的ADD指令。而且，其后的指令没有依赖它的执行结果。因此，这条指令可以跟着循环末的转移指令移入延迟槽，如图8-22b所示。因为不管转移的结果如何，它都可以被执行，所以转移指令中的无效位设置为0（这是默认条件）。

	LDX	R3, 0, R6	载入列表的项数
	OR	R0, R0, R4	R4作为列表中的偏移量
	OR	R0, R0, R7	清除R7作为累加器
LOOPSTART	LDX	R3, R4, R5	将列表项载入R5
	ADD	R5, R7, R7	将数加到累加器中
	ADD	R4, 8, R4	指向另一入口
	SUBcc	R6, 1, R6	减小R6并设置状态标志
	BG	xcc, LOOPSTART	如果列表中还有表项则继续循环
NEXT	...		

a) 所要求的循环程序

	LDX	R3, 0, R6	
	OR	R0, R0, R4	
	OR	R0, R0, R7	
LOOPSTART	LDX	R3, R4, R5	
	ADD	R4, 8, R4	
	SUBcc	R6, 1, R6	
	BG,pt	xcc, LOOPSTART	采用预测，无效位为0
	ADD	R5, R7, R7	
NEXT	...		

b) 使用延迟槽重新组织的指令

图8-22 展示转移延迟槽和转移预测作用的加法循环

至于转移预测,我们应该注意到,循环执行次数等于表项的数目。这意味着,除了 $n = 1$ 的情形,在退出循环前转移会发生多次。因此,我们在BG指令中设置转移预测位以表明期望进行转移。

并不是只有条件转移指令才能检查条件码标志。例如,条件传送指令MOVcc,只要条件码满足指令后缀cc规定的条件,该指令就将数据从一个寄存器拷贝到另一个寄存器中。考虑这样的两条指令

```
CMP R5, R6
MOVle icc, R5, R6
```

这里如果icc中的条件码标志小于或等于条件 ( $Z + (N \oplus V) = 1$ ),那么MOVle指令将R5的内容拷贝到R6中。最后的结果是将两个数中较小的一个放入寄存器R6。没有条件传送指令时,完成同样的任务时需要一个转移指令,指令顺序如下:

```
CMP R5, R6
BG icc, GREATER
MOVA icc, R5, R6
```

GREATER ...

其中MOVA是“一直传送”(move-always)指令。MOVle指令不仅减少了所需要的指令数量,而且更重要的是它避免了在流水线执行中由转移指令引发的性能下降。

该指令集还有很多其他的一些特征,这些特征可以最大限度提高流水线超标量处理器的性能。我们将在UltraSPARC II 处理器的上下文中讨论这些特征。这些特征包含的思想在本章中已经介绍过。

## 8.7.2 UltraSPARC II

UltraSPARC II 处理器的主要构件块如图8-23所示。该处理器使用两级高速缓存:一个外部高速缓存(E-cache)和两个内部高速缓存,而在两个内部高速缓存中,一个用于指令(I-cache),一个用于数据(D-cache)。外部的高速缓存控制器在处理器芯片上,用于存储器管理的控制硬件也在处理器芯片上。存储器管理部件使用两个转换缓冲器,一个用于指令iTLB,一个用于数据dTLB。处理器用系统内部互连总线与存储器和I/O子系统进行通信。

有两个执行部件,一个用于整数操作,一个用于浮点操作。每个部件中包含一个寄存器集以及两个用于指令执行的独立流水线。因此,处理器能够同时执行四条指令:两条整数指令和两条浮点数指令。这些并行执行的指令每条都经过各自的流水线。如果有足够的指令并且四条流水线中没有延迟,那么每个时钟周期都有四条新指令进入执行阶段。

处理器的预取和调度部件(PDU)负责为执行部件供给连续的指令。PDU是在指令需要之前预取指令,并把它们放在称为指令缓冲器的临时存储器中,指令缓冲器完成图8-19中指令队列的作用。

## 8.7.3 流水线结构

UltraSPARC II 有一条9段指令执行流水线,如图8-24所示。每一段的功能在一个处理器时钟周期内完成。我们先给出流水线操作的总体介绍,之后再具体讨论每一阶段。

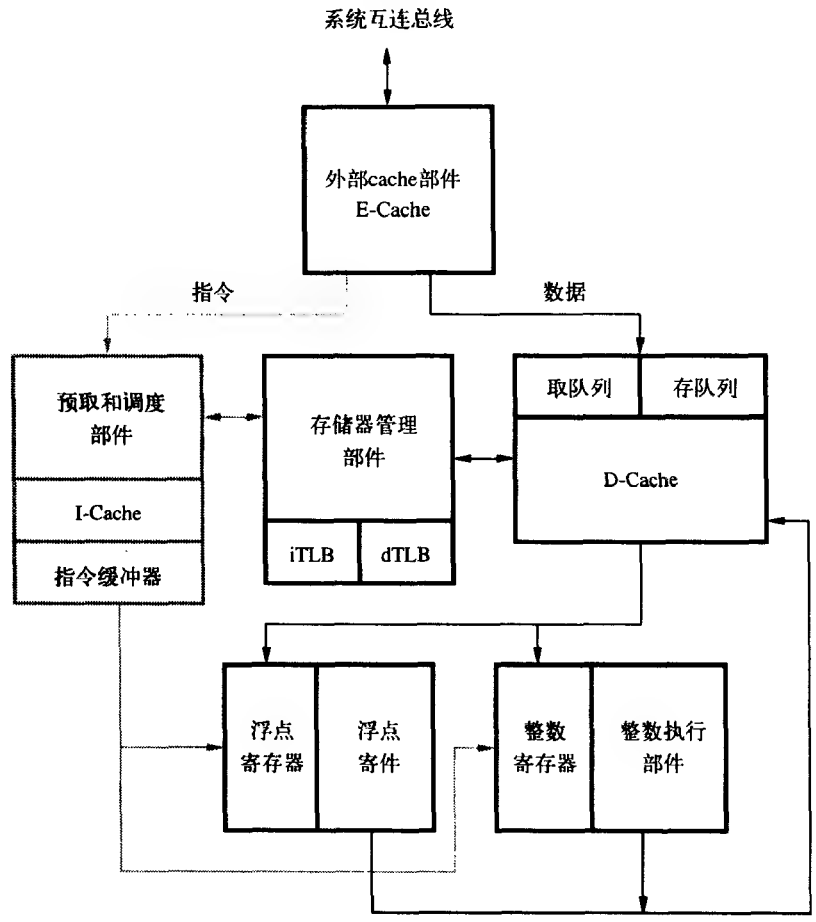


图8-23 UltraSPARC II 处理器的主要构件块

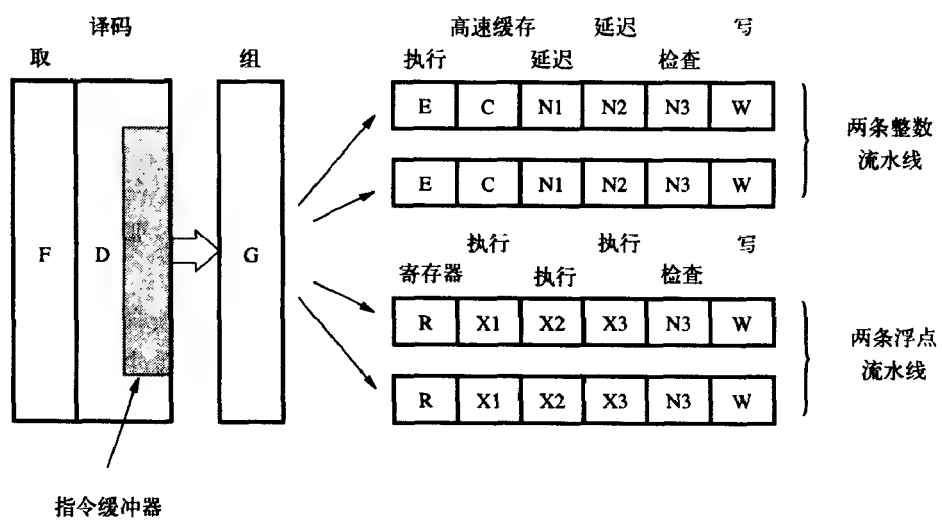


图8-24 UltraSPARC II 处理器的流水线组织结构

流水线的前三段对所有指令是公用的。在第1段（F）从指令高速缓存中取指令，在第2段

(D)进行部分译码。接着在第3段(G)选择一组最多四条指令并行执行。这些指令接着被调度到整数和浮点执行部件中。

两个执行部件中每个都包含两个6段的并行流水线。前四段执行指令指定的操作,后两段用于检查异常以及保存指令结果。

#### 指令读取和译码

PDU从指令高速缓存中取四条指令进行部分译码,并将结果存在指令缓冲器中。指令缓冲器最多可以容纳12条的指令。这一阶段的译码使PDU能够判定指令是否是转移指令。它还检测那些可以对流水线中的指令加速决策的突出特性。

指令高速缓存中的一个高速缓存块由32个字节构成。一个块可以容纳8条指令。指令被装入高速缓存时是按照虚拟地址存储的,以便PDU可以迅速地读取指令而无需地址转换。只要每组指令都不穿越高速缓存块的边界,PDU可保持每周周期四条指令的读取速度。如果高速缓存块中剩下不到四条指令,PDU就只读取当前块中剩余的指令。

PDU使用4状态转移预测算法,与图8-15的描述类似。它利用转移指令中的转移预测位将初始状态设置为LT或LNT。对于高速缓存中的每两条指令,PDU用两位来记录转移预测算法的状态。这些位存储在高速缓存与该指令相关的标志中。

495

对于指令高速缓存中的每四条指令,提供了称为Next Address的标志字段。当执行最先取出的指令时,PDU计算转移指令的目标地址,并将该地址记录在Next Address字段中。这个字段使得在以后的步骤中不必每次重新计算目标地址,就可以继续预取指令。由于每半条高速缓存线只有一个Next Address字段,所以只有在每组的四条指令中最多有一条转移指令的情况下,它的优点才能充分体现出来。

#### 分组

在流水线的第3段G中,分组逻辑选择要并行执行的一组最多四条指令,将它们调度到整数和浮点执行部件中。图8-25所示为一个简短的指令序列以及这些指令要调度的方法。该图的b部分和c部分是分别表示PDU预测发生转移和不发生转移时的指令分组情况。注意,延迟槽中的指令FCMP在这两种情况下都包括在选定的组中。该指令将被执行,但直到做出转移决策时才会提交。如果发生转移,FCMP的结果就是无效的,因为转移指令的无效位被置为1。每组的前两条指令调度到整数部件,后两条指令调度到浮点部件。

分组逻辑电路负责确保它调度的指令是已经准备就绪的指令。例如,在一个组中指令使用的所有操作数必须都已经准备好。如果一条指令依赖另一条指令的结果,那么这两条指令不能分在同一组中。转移指令不包含在这个条件中,下面将简要解释。

指令按程序的顺序调度。回想一下如果分组中包括转移指令,由于预取和译码部件的转移预测,转移指令就会已经尝试着被执行了。因此,基于这种预测,指令缓冲器中的指令就会以正确的次序执行。分组逻辑按顺序对指令缓冲器中的指令做简单检查,目的是在队首选择出满足分组约束的最大数。

分组逻辑在挑选要包括在分组中的指令时,要考虑的一些约束有:

1. 指令只能按顺序调度。如果一条指令不包括在组中,那么其后的所有指令都不能被选择。
2. 同一组中,一条指令的源操作数不能依赖于任何其他指令的目标操作数。这条规则有两项例外:

	ADDcc	R3, R4, R7	$R7 \leftarrow [R3] + [R4]$ 设置条件码
	BRZ,a	Label	如果为0转移, 将取消位置1
	FCMP	F1, F5	FP: 比较[F2]和[F5]
	FADD	F2, F3, F6	FP: $F6 \leftarrow [F2] + [F3]$
	FMOVs	F3, F4	将单精度操作数从F3移动到 F4
	⋮		
Label	FSUB	F2, F3, F6	FP: $R6 \leftarrow [F2] - [F3]$
	LDSW	R3, R4, R7	将[R3] + [R4]处的单字载入到R7
	⋮		

a) 程序片断

	ADDcc	R3, R4, R7
	BRZ,a	Label
	FCMP	F1, F5
	FSUB	F2, F3, F6

b) 发生转移时的指令分组

	ADDcc	R3, R4, R7
	BRZ,a	Label
	FCMP	R1, R5
	FADD	R2, R3, R6

c) 不发生转移时的指令分组

图8-25 指令分组实例

496

• 将寄存器的内容存到存储器的存储指令可以与前面使用该寄存器作为目标寄存器的指令分为一组。这是允许的, 因为流水线直到以后存储指令阶段时才需要数据。下面会看到这种情形。

• 转移指令可能与前面设置条件码的指令分为一组。

3. 同一组中的两条指令不能有相同的目标操作数, 除非目标操作数是寄存器R0。例如, 图8-26a中的LDSW指令不能与ADD指令分为一组, 应延迟到下一组中, 如图所示。

4. 在一些情形下, 某些指令相对于其他指令必须延迟两到三个时钟周期。例如, 条件指令

MOVrz R1, R6, R7

(根据寄存器的情况传送) 如果R1的内容等于0, 就将R6的内容传送到R7。这条指令需要一个额外的时钟周期来检查R1的内容是否等于0。因此, 读寄存器R7的指令不能分在同一组或随后的组中。此类指令的最早调度如图8-26b所示。

当分组逻辑将指令调度到整数部件时, 它也同时从整数寄存器文件中取出那条指令的源操作数。访问寄存器文件所需要的信息在译码位中可以得到。译码位由预取译码部件送入指令缓冲器。所以, 到G段的时钟周期末, 有一条或两条整数指令准备就绪可进入执行阶段。从寄存器文件读到的数据存入中间存储缓冲器, 如图8-27所示。指令传送到浮点部件后, 在R段存取浮点寄存器文件中的操作数。

ADD	R3, R5, R6	G	E	C	N1	N2	N3	W
LDSW	R4, R7, R6	G	E	C	N1	N2	N3	W

a) 有公共目标操作数的指令

MOVRZ	R1, R6, R7	G	E	C	N1	N2	N3	W				
OR	R7, R8, R9			G	E	C	N1	N2	N3	W		

b) 由MOVR指令引起的延迟

图8-26 由于阻塞的调度延迟

497

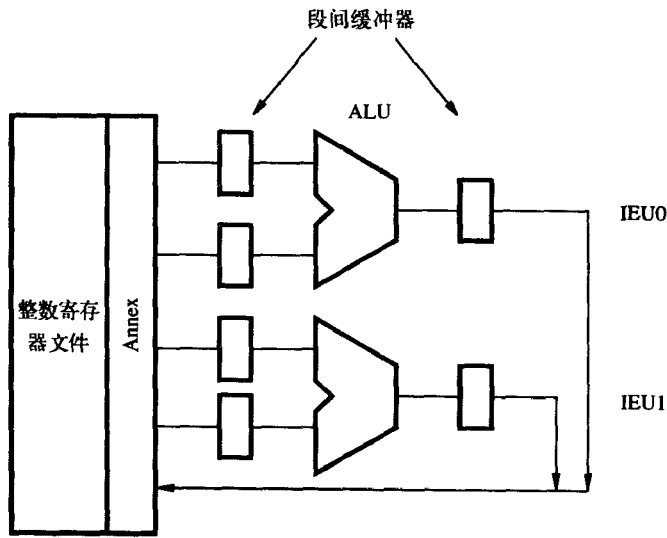


图8-27 整数执行部件

### 执行部件

整数执行部件包括两个相似但不同的部件IEU0和IEU1。只有部件IEU0能够处理移位指令，同时只有IEU1能够产生条件码。不涉及这两种操作的指令可以在任何一个部件上执行。

大多数整数指令的ALU操作都在一个时钟周期内完成，即流水线的E阶段。在这个时钟周期末，将结果保存在如图8-27所示的ALU输出端的缓冲器中。在下一个时钟周期C段，该缓冲器的内容传送到寄存器文件的Annex部分中。Annex中包含着用于寄存器重命名的临时寄存器，这在8.6节阐述过。在流水线的W段，临时寄存器的内容传送到相应的永久寄存器中。

在C段的另一个动作是产生条件码。当然，只有指定要设置条件码标志位的指令，例如ADDcc，才会产生条件码。这类指令必须在IEU1部件中执行。

498

考虑设置条件码标志的Icc指令以及检查这些标志位的后续条件转移指令BRcc。当预取和调度部件遇到BRcc指令时，可能还没有得到Icc执行的结果。PDU预测转移结果在此基础上继续预取指令。稍后，当Icc到达流水线的C段时产生条件码，并且在同一时钟周期将条件码送到PDU。PDU检查它的转移预测是否正确。如果正确，继续执行而不用中断。否则，清除流水线和指令缓冲器的内容，PDU重新取出正确指令。因为这些指令还未到达流水线的W段，所以可以在此

处清除。

当转移预测不正确时,可能已错误地预取并部分地执行了许多指令。这种情形如图8-28所示。我们假定分组逻辑在三个连续时钟周期调度四条指令。指令Icc在第一组的开始设置了条件码,该条件码由随后的BRcc指令测试。当第一组到达C段时进行该测试。这时,第三组I<sub>9</sub>到I<sub>12</sub>进入流水线的G段。如果转移预测不正确,那么将取消I<sub>4</sub>到I<sub>12</sub>的九条指令(注意延迟槽中的指令I<sub>3</sub>总是被执行的)。此外,可能已经预取并装入指令缓冲器中的其他指令也要被丢弃。因此,在极限情形下,可能会丢弃21条指令。

在流水线的N1和N2段,不能执行任何操作。所以在这两个段引入两个时钟周期的延迟,使得整数流水线的整个长度与浮点流水线的长度相同。对于在C段没有执行完的整数指令,例如除法指令,在N1和N2段继续执行。如果需要更多的时间,则在N1和N2之间插入额外时钟周期。指令只有在执行的最后一个时钟周期才进入N2。例如,如果执行一条指令操作需要16个时钟周期,则需要N1段之后插入12个时钟周期。

浮点执行部件也有两个独立的流水线。在R段读取寄存器操作数,该操作经历三个流水线段(X1到X3)完成。这里如果需要额外的时钟周期,例如求平方根指令,要在X2和X3之间插入额外的时钟周期。

在N3段,处理器检查各种异常条件来判定是否会发生陷阱(中断)。最后,在写入段(W),指令的结果保存在目标单元(一个寄存器或数据高速缓存)中。在进入此段之前的任何时刻,指令都可能被清除,并且指令的所有作用都被取消。一旦进入写阶段,指令的执行就不能停止了。

#### 读取和存储部件

##### 指令

LDUW R5, R6, R7

从存储器的[R5]+[R6]单元处将32位无符号字取到寄存器R7中。对于其他整数指令,在流水线的G段期间读取寄存器R5和R6的内容。然而,指令和指令操作数并不是送入其中一个整数执行部件中,而是转发到读取和存储部件中,如图8-29所示。在E段,该部件通过把寄存器R5和R6的内容相加来产生要访问的存储单元的有效地址。这个结果是一个虚拟地址值,它被送入数据高速缓存中,同时送入数据转换缓冲器dTLB中准备转换成物理地址。

数据按虚拟地址存入高速缓存中,因而无须等待地址转换结束就可以迅速访问数据。在C段,数据和相应的标志信息都从D-Cache中读出,物理地址从dTLB中读出。D-Cache中用的标志是物理地址数据的一部分。在N1段,从D-Cache中读出的标志与从dTLB得到的物理地址相核对。命中时,数据取入Annex寄存器中,在W段送到目标寄存器。如果标志不匹配,指令进入读取/存储队列,在那里等待从外部高速缓存读入D-Cache中的高速缓存块。

一旦指令进入读取/存储队列,就不再认为它是在执行流水线中了。当一条装入指令正在队列中等待时,其他指令可以继续执行直到完成,除非其中某条指令引用了正等待存储器数据的

I <sub>1</sub> (Icc)	G	E	C
I <sub>2</sub> (BRcc)	G	E	C
I <sub>3</sub>	G	E	C
I <sub>4</sub>	G	E	C
I <sub>5</sub>		G	E
I <sub>6</sub>		G	E
I <sub>7</sub>		G	E
I <sub>8</sub>		G	E
I <sub>9</sub>			G
I <sub>10</sub>			G
I <sub>11</sub>			G
I <sub>12</sub>			G

↑ 中止

图8-28 错误转移预测最差情形下的时序

499

500



寄存器（上例的R7）。因此，读取/存储队列使流水线操作从外部数据存取操作中分离出来，以便这两个操作可以独立执行。

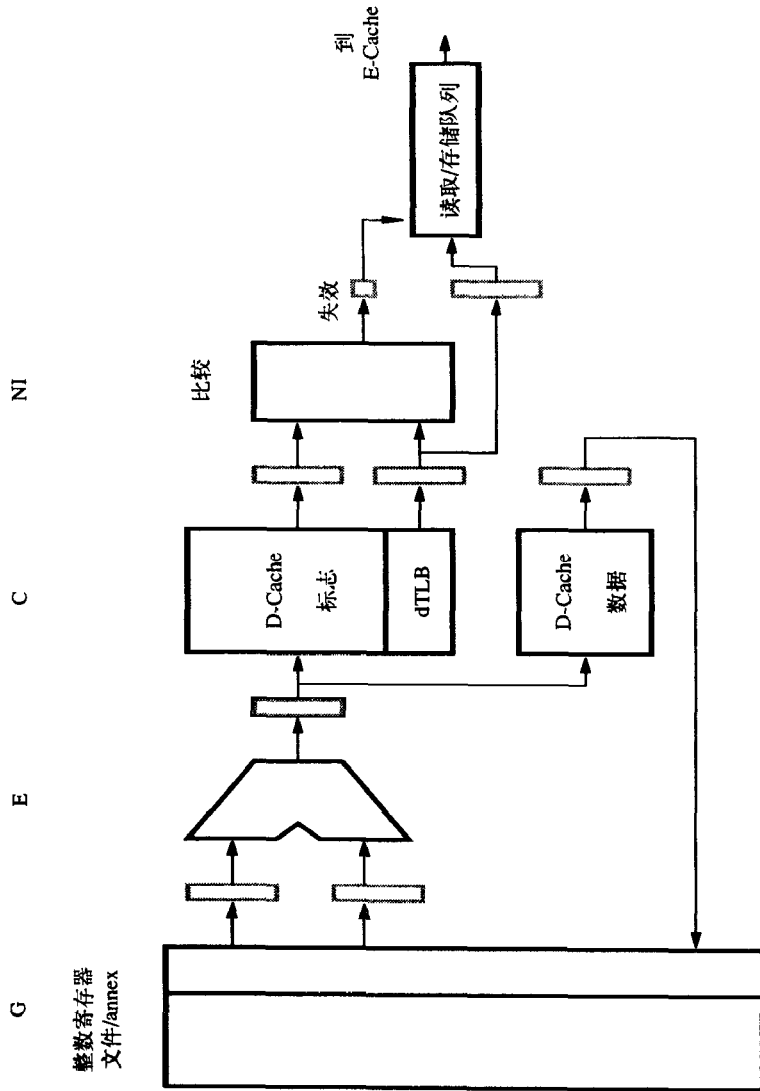


图8-29 读取和存储部件

### 执行流程

分析UltraSPARC II 处理器中以及UltraSPARC II 处理器与外部高速缓存和存储器之间的指令和数据流是有意义的。图8-30是图8-23重新组织后的主要功能部件，它用来说明指令流和数据流，以及指令和数据队列所起的作用。

指令从I-Cache中取出并装到指令缓冲器中，指令缓冲器可以存储12条指令。从那里，每次将四条指令传送到称为“内部寄存器和执行部件”的模块中，指令在那里被执行。一般，PDU填满指令缓冲器的速度高于分组逻辑调度指令的速度。因此，指令缓冲器大部分时间是满的。没有高速缓存失效及转移预测错误时，内部执行部件永远不会缺乏指令。类似地，在大部分时间里，读取和存储指令的内存操作数可能在数据高速缓存中找到，所以用一个时钟周期就可以

进行存取操作。因此执行可以正常进行而没有延迟。

当指令高速缓存失效时, 从外部高速缓存中读取正确的块需要几个时钟周期的延迟。这期间, 分组逻辑继续从指令缓冲器中调度指令直到缓冲器为空。根据处理器模型不同, 可能要用三或四个时钟周期从外部高速缓存中读取高速缓存块(八条指令)。这个时间与分组逻辑在满指令的缓冲器中调度指令所用的时间基本相同(注意在每个时钟周期并不是总能调度四条指令)。因此, 如果发生高速缓存失效时指令缓冲器是满的, 流水线的执行操作可能根本不会中断。如果外部高速缓存也发生失效, 就需要更多的时间来访问存储器。在这种情形下, 难免会发生流水线的拖延。

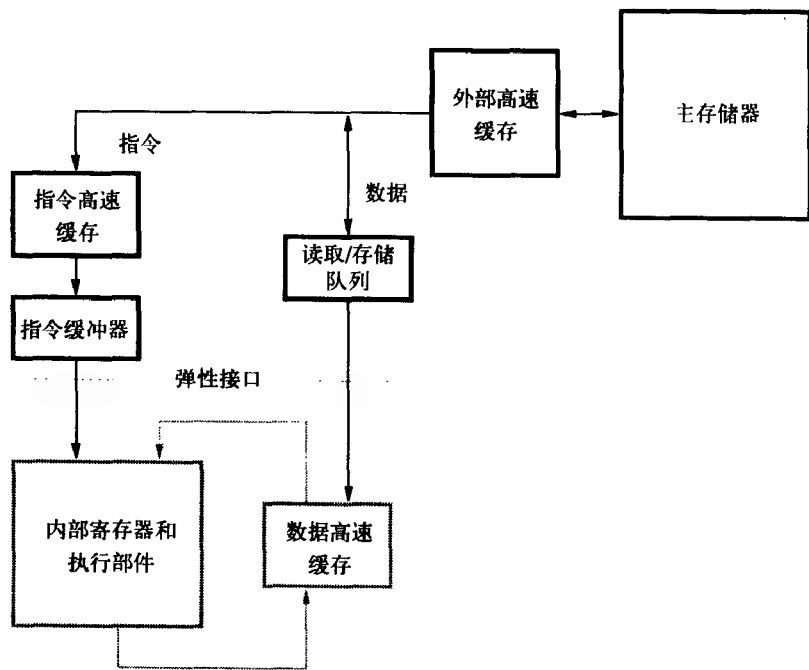


图8-30 执行流程

造成高速缓存失效的取操作进入存取队列并等待从外部高速缓存或存储器中传送数据。然而, 只要取操作的目标寄存器没有被后面的指令引用, 内部指令就可以继续执行。因此, 指令缓冲器和存取队列把内部处理器流水线从外部数据传输中隔离出来。它们作为一种弹性接口, 在慢速外部数据传输的同时允许继续运行内部的高速流水线。

## 8.8 性能考虑

我们在1.6节指出过拥有动态指令数 $N$ 的程序执行时间 $T$ 由公式

$$T = \frac{N \times S}{R}$$

给出, 其中 $S$ 是读取和执行一条指令所用时钟周期的平均数,  $R$ 是时钟频率。这个简单模型假设指令是按照一条接着一条执行的, 中间没有重叠。一个有用的性能指标是指令吞吐量, 它是每秒执行指令的数量。对于顺序执行, 吞吐量由

$$P_s = R/S$$

给出。

本节中，我们来了解流水线对指令吞吐量的提高程度。不过应再强调一点在第1章中提出的关于性能的度量。度量性能的惟一实际量是程序的整个执行时间，如果指定的任务需要执行大量的指令，那么较高的指令吞吐量也不一定会带来高性能。由于这个原因，在比较两个处理器时，第1章描述的SPEC速率提供了一个较好的衡量指标。

图8-2说明4段流水线可以使指令吞吐量提高4倍。通常， $n$ 段流水线可以将吞吐量提高 $n$ 倍。因此，似乎 $n$ 值越高，得到的性能就越强。这会引出两个问题：

503

- 指令吞吐量的潜在提高实际上能实现多少？
- $n$ 值最好为多少？

只要流水线出现阻塞，指令吞吐量就会降低。因此，流水线的性能大大受到诸如转移和高速缓存失效代价等因素的影响。首先，我们讨论这些因素对性能的影响，然后讨论应该采用多少流水线段的问题。

### 8.8.1 指令阻塞的影响

在前几节已经定性地介绍了各种阻塞的影响。现在我们定量地评估一下高速缓存失效和转移代价的影响。考虑一个使用图8-2中的4段流水线的处理器。时钟频率，也就是分配给流水线中每段的时间，由最长的段决定。设通过ALU的延迟为临界参数。它是两个整数相加所需要的时间。因此，如果ALU延迟是2ns，可使用500MHz时钟。这个处理器片上的指令和数据高速缓存也应该将它们的存取时间设计为2ns。在理想条件下，这个流水线处理器的指令吞吐量 $P_p$ 由

$$P_p = R = 500\text{MIPS} \quad (\text{每秒百万指令数})$$

给出。

为了评价高速缓存失效的影响，我们使用与在5.6.2节同样的参数。在该系统中高速缓存失效代价 $M_p$ 计算出来为17个时钟周期。设 $T_1$ 是完成两条连续指令之间的时间。对于顺序执行， $T_1 = S$ 。然而，在没有阻塞时，流水线处理器每时钟周期执行完一条指令，因此 $T_1 = 1$ 周期。高速缓存失效使流水线延迟的时间等于高速缓存失效的代价。这意味着 $T_1$ 增加的数量值等于高速缓存失效时指令的高速缓存失效代价。发生高速缓存失效可能是由指令或数据引起的。考虑一台指令和数据共享高速缓存的计算机，设 $d$ 是与内存操作数有关的指令的百分比。由于高速缓存失效使 $T_1$ 的值平均提高的结果由下式给出：

$$\delta_{\text{miss}} = ((1 - h_i) + d(1 - h_d)) \times M_p$$

其中 $h_i$ 和 $h_d$ 分别是指令和数据的命中率。假设30%的指令访问存储器中的数据。当指令命中率为95%，数据命中率为90%时，

$$\delta_{\text{miss}} = (0.05 + 0.3 \times 0.1) \times 17 = 1.36 \text{ 周期}$$

504

将这个延迟考虑进来，那么处理器的吞吐量为

$$P_p = \frac{R}{T_1} = \frac{R}{1 + \delta_{\text{miss}}} = 0.42R$$

注意， $R$ 用MHz表示，直接得到的吞吐量是每秒百万条指令。当 $R = 500\text{MHz}$ 时， $P_p = 210\text{MIPS}$ 。

将这个值与没有流水线得到的吞吐量相比较。使用顺序执行的处理器每条指令需要4个周期。它的吞吐量为

$$P_s = \frac{R}{4 + \delta_{miss}} = 0.19R$$

当  $R = 500\text{MHz}$  时,  $P_s = 95\text{MIPS}$ 。很明显大大提高了流水线的吞吐量。但是性能增益  $0.42/0.19 = 2.2$ , 只是理想情形的一半多一点。

在流水线处理器中降低高速缓存失效代价尤其重要。正如在第5章中阐述的, 这可以通过在片上高速缓存和存储器之间引入第二级高速缓存来实现。假设从第二级高速缓存传送一个8个字的块需要的时间是  $10\text{ns}$ 。因此, 如果主高速缓存失效, 在第二级高速缓存中存在所需要块时, 将会引入5个周期的代价  $M_s$ 。在第二级高速缓存失效的情形下, 会带来总共17个周期的代价 ( $M_p$ )。因此, 假设第二级高速缓存的命中率为94%,  $T_1$  平均提高

$$\delta_{miss} = ((1 - h_i) + d(1 - h_d)) \times (h_i \times M_i + (1 - h_i) \times M_p) = 0.46 \text{ 周期}$$

这种情形下的指令吞吐量为  $0.68R$  或  $340\text{MIPS}$ 。等价的非流水线处理器吞吐量为  $0.22R$  或  $110\text{MIPS}$ 。因此, 流水线提供的性能增益为  $0.68/0.22 = 3.1$ 。

事实上, 1.36和0.46的值是有点悲观, 因为我们假设每次发生数据失效时会带来全部的失效代价。而这只是当处理器等待存储器访问结束时, 紧跟在访问存储器指令之后的那条指令被延迟的情形。然而, 优化编译器会尽力在产生依赖性的两条指令之间放入其他指令来增加距离。而且, 在使用指令队列的处理器中, 在取指令期间, 由于处理器能够从队列中调度指令, 所以高速缓存失效代价的影响会大大降低。

## 8.8.2 流水线的段数

$n$ 段流水线可以使指令吞吐量提高  $n$  倍的事实促使我们使用大量的流水线段。然而, 随着流水线段数的增加, 流水线延迟的概率也会相应增加, 因为会有更多的指令并发地执行。因此, 相距很远的指令之间的依赖性还可能会引起流水线的延迟。而且, 转移代价可能会变得更大, 如图8-9所示。由于这些原因,  $n$ 值增加所带来的增益开始减小, 相关的开销也就不再合理了。

另一个重要因素是处理器执行中基本操作的固有延迟, 其中最重要的是ALU延迟。在许多处理器中, 处理器将完成一项ALU操作的时间当作一个时钟周期的大小。其他操作按照加法操作所用的时间分成几步。对于使用流水线的ALU这种方法也是可行的。例如, Compaq Alpha 21064处理器的ALU包括2段流水线, 每段用5ns完成它的操作。

许多流水线处理器使用4到6个段。另外一些处理器将指令执行分成更小的段, 使用更多的流水线段数以及更快的时钟。例如, UltraSPARC II使用9段流水线, Intel的PentiumPro使用12段流水线。最新的Intel处理器Pentium4有一条20段的流水线, 使用的时钟频率范围在1.3~1.5GHz。为了快速操作, 一个时钟周期内可以存在两个流水线段。

## 8.9 结束语

本章介绍了两个重要特性: 流水线和指令多发。流水线使我们能够设计出指令吞吐量达到每个时钟周期一条指令的处理器。多发使得指令吞吐量为每个时钟周期有多条指令的超标量操作成为可能。

只有认真解决以下三个方面的问题, 才可以实现性能的提高:

- 处理器的指令集
- 流水线硬件的设计
- 相关编译器的设计

认识到三者之间的相互作用是非常重要的。在一个处理器的设计过程中, 由于对这三者之间相互作用的认识程度不同, 设计出来的处理器性能水平也不同。重视程度越高, 处理器的性能水平就越高。对于现代处理器来说, 特别适用流水线执行的指令集是它的关键特征。

## 习题

### 8.1 考虑下列指令序列

```
Add #20, R0, R1
Mul #3, R2, R3
And #$3A, R2, R4
Add R0, R2, R5
```

在所有指令中, 目标操作数在最后给出。寄存器R0和R2的初始值分别为2000和50。这些指令在包含有类似于图8-2所示的4段流水线上执行。假设在第1个时钟周期取第1条指令, 并且取第1条指令只需1个时钟周期。

(a) 画出类似于图8-2a的图。描述在第1到第4个时钟周期的每一个周期中流水线中每一段实现的操作。

(b) 写出第2到第5时钟周期中中间存储缓冲器B1、B2和B3的内容。

### 8.2 针对下列程序

```
Add #20, R0, R1
Mul #3, R2, R3
And #$3A, R1, R4
Add R0, R2, R5
```

重新回答8.1中的问题。

- 8.3 图8-6中的指令 $I_2$ 由于依赖于 $I_1$ 的结果而延迟。通过占用译码段, 指令 $I_2$ 阻塞 $I_3$ ,  $I_3$ 进而又阻塞 $I_4$ 。假设 $I_3$ 和 $I_4$ 不依赖于 $I_1$ 或 $I_2$ , 并且寄存器文件允许两步写并行进行, 如何利用附加的存储缓冲器使 $I_3$ 和 $I_4$ 比图8-6中所示早一些进行处理? 重新画图, 指出各步的新次序。
- 8.4 图8-6中产生延迟气泡是因为指令 $I_2$ 在译码段延迟, 由此导致了指令 $I_3$ 和 $I_4$ 被延迟, 即使 $I_3$ 和 $I_4$ 不依赖于 $I_1$ 或 $I_2$ 。假设译码段允许两步译码并行进行。证明如果寄存器文件也允许两个写步骤并行进行, 可以完全消除延迟气泡。
- 8.5 图8-4给出了一条指令因高速缓存失效而引起的延迟。用图8-10的硬件结构重画此图。假设指令队列可容纳四条指令, 并且取指令部件一次从高速缓存中读两条指令。
- 8.6 一个循环程序以一条转移到循环开始的条件转移语句结束。如何在使用包含有延迟槽的延迟转移的流水线计算机上实现这个循环? 在什么条件下能够将有效指令放到延迟槽中?
- 8.7 UltraSPARC II处理器的转移指令有一个无效位。当被编译器设置时, 如果不发生转移就丢弃延迟槽中的指令。一种替代方法是如果发生转移就丢弃该指令。何时选择使用这两种方法?

- 8.8 一台计算机有一个延迟槽。延迟槽中的指令总会被执行，但这只是在纯理论的基础上。如果转移不发生，这条指令的结果被丢弃。提出一种在这台计算机上有效地实现循环程序的方法。
- 8.9 为SPARC处理器重写图2-34的排序程序。注意SPARC体系结构有一个含有相关无效位的延迟槽并使用转移预测。尽量用有效指令填充延迟槽。
- 8.10 考虑一个语句格式

IF  $A > B$  THEN 动作1 ELSE 动作2

的描述。写一个汇编语言指令序列，首先只使用转移指令，接着使用如同在ARM处理器上提供的那些条件指令。假设使用一条简单的2段流水线，画出类似于图8-8的图用来比较两种方法的执行时间。

507

- 8.11 图8-7中的前馈通路（粗线）允许RSLT寄存器的内容直接在ALU中使用。运算的结果存回到RSLT寄存器中，并替换它先前的内容。使用什么类型的寄存器可以使这种运算成为可能？  
考虑两条指令

$I_1$ : Add R1,R2,R3

$I_2$ : Shift\_left R3

假设在指令 $I_1$ 执行前，R1、R2、R3和RSLT的值分别为30、100、45和198。画出一条4段流水线的时序图，在每个周期标出时钟信号和RSLT寄存器的内容。用你所画的图证明转发操作可以得到正确的结果。

- 8.12 为一个处理器写一个如图2-37的程序，在该处理器中只有读取和存储指令可以访问存储器。标出程序中的所有依赖性，并指出如何优化该程序以便在流水线处理器上执行。
- 8.13 假设在计算机上执行的动态指令数的20%是转移指令。使用包含有一个延迟槽的延迟转移估计出性能的增益，假设编译器能够使用延迟槽的85%。
- 8.14 一个流水线处理器包含有两个转移延迟槽。一个优化编译器能够占满其中一个延迟槽85%的时间，另一个只有20%。这种优化使性能提高的百分比是多少？假设所执行指令的20%是转移指令。
- 8.15 一个流水线处理器使用延迟转移技术。请你从这种处理器的两种可能设计中推荐一种。在第一种可能性中，处理器拥有一条4段流水线和—个延迟槽，在第二种可能性中，它拥有一条6段流水线和两个延迟槽。比较这两种方案的性能，只考虑转移代价。假设20%的指令是转移指令并且优化编译器填满单个延迟槽的成功率是80%。对于第二种方案，编译器能填满第二个延迟槽25%的时间。
- 8.16 考虑一个使用图8-15b表示的转移预测机制的处理器。初始状态是LT或LNT，这取决于转移指令所提供的信息。讨论编译器应如何处理用于控制“do while”和“do until”循环的转移指令，并讨论每一种情形下转移预测机制的适用性。
- 8.17 假设图8-10的指令队列可容纳六条指令。重画图8-11，假设队列在时钟周期1是满的，并且取指令部件一次能从高速缓存中读两条指令。在指令 $I_k$ 被取出后，队列什么时候再次被填满？
- 8.18 针对图8-14中转移预测错误的情形重画图8-11。
- 8.19 图8-16给出了使用复合寻址方式的指令用相同的时间来执行使用简单寻址方式的一个等价

508

指令序列。然而，使用简单寻址方式是RISC设计思想的一个原则。你如何设计一条流水线来处理复合寻址方式？讨论这种方法的利弊。

### 参考文献

1. The SPARC Architecture Manual, Version 9, D. Weaver and T. Germond, ed., PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

### 本章目标

在本章中你将学习以下内容:

- 嵌入式应用
- 嵌入式系统中的微控制器
- 处理I/O设备限制问题
- 使用C语言控制I/O设备
- 片上系统设计

511

计算机系统被用在了各种应用中,因此它们可表现出不同的结构、不同的规模和不同的性能。对于任何一个给定的应用必须考虑的重要因素包括性能、可靠性和成本。任何一个可以处理大量图形信息,包括图片图像和动画的计算机,都必须拥有良好的实时性。对于个人计算机和 workstation 来讲,重要的是在满足市场价格要求的情况下达到最好的性能。特别高的性能必定意味着价格昂贵,这种机器对那些要求在给定的时间里要完成大量计算的问题是需要的。计算性要求高的应用例子有:复杂系统的仿真、在一个印刷电路板上确定一个好的布线方案以及许多的计算机辅助设计(CAD)任务。这类应用在个人计算机上运行可能要花费许多小时才能完成。通常需要将它们放在一个计算服务器上运行,相对高价格的服务器可以实现非常高的性能。第8章中的许多内容涉及到建立高性能计算机的问题。

在许多应用中不需要高性能的处理器。目前微处理器控制广泛地应用在照相机、移动电话、可视电话、销售终端、厨房器具、汽车和许多玩具中。在这些应用中高性能不是主要的问题,低费用和高可靠性是主要的需求,小型化和低功耗通常是最重要的问题。所有这些应用可以用一个芯片去完成,在这个芯片中不仅包含处理器而且还包含一些输入/输出接口、计时器电路并具有一些其他的设计特性。这些设计特性是指使用少量的芯片就能容易地完成一个完整的计算机控制系统。这些包含I/O接口和一些存储器的微处理器芯片通常称为微控制器。用于对特定目标进行计算机控制而不是用于通用计算的计算机系统,称为嵌入式系统。这种系统是本章讨论的主题。

### 9.1 嵌入式系统的实例

在本节中我们给出三个嵌入式系统的实例,说明在一个典型的嵌入式应用中所需要的处理过程和控制功能。



### 9.1.1 微波炉

许多家用电器使用计算机控制去管理它们的操作,典型的例子就是微波炉。这种用具基于一个磁控管发生器产生的微波去加热在一个封闭空间中的食物。当微波炉开启时,磁控管产生最大的功率输出,使用依据受控的时间间隔打开或关闭磁控管的方式达到降低功率的目的。这样,用控制功率和总加热时间的方法,可以实现烹调操作中各种用户的选择。

微波炉的具体规格说明可能包括以下的烹调操作:

- 手动选择功率级别和烹调时间。
- 手动选择不同烹调步骤的次序。
- 在用户指明了食物类型(例如:肉、蔬菜或爆米花)以及食物的重量时自动选择。一个合适的功率和时间随后由控制器计算出来。
- 根据指定食物重量的自动解冻。

炉中包含一个显示输出,它可以显示:

- 每日时钟定时。
- 烹饪过程中时钟定时器递减。
- 给用户的提示信息。

以蜂鸣声形式发出的声音报警信号,用于表示烹饪操作结束。还需要提供一个排风扇和一个炉灯信号。最后,如果炉子的门被打开,门锁必须关闭磁控管。所有这些功能可以用微处理器控制。

与用户有关的输入/输出功能必须包括:

- 输入键,它包括0到9的数字键以及Reset(重置)、Start(开启)、Stop(暂停)、Power Level(功率级别)、Auto Defrost(自动解冻)、Auto Cooking(自动烹饪)、Clock Set(时钟设置)及Fan Control(风扇控制)这样的功能键。为了减少键的总数量,有些键可能有多重功能。例如,按下Fan Control键一段时间可以用来选择风扇的速度。
- 液晶显示器形式的显示输出(类似于七段显示,在A.9节中讨论)。
- 一个小型扬声器用来产生蜂鸣声。

用于微波炉的微控制器可以用一个小型的微处理器——基本计算机单元去实现。因为这里的计算任务非常简单,仅包括维护每天的时钟时间、确定在各种烹饪操作中所需的动作、产生打开或关闭磁控管和电风扇这类设备的控制信号以及生成显示信息。因此可以使用相对简单的处理器去执行这些任务。实现所需动作的程序非常小,它们必须存储在一个不易丢失的只读存储器中,这样当关闭电源时才不会丢失。还需要有一个小型的RAM用于在运算过程中执行和保存用户的输入数据。最重要的需求是要具备大量的I/O能力使其能够与所有的输入键、显示器和输出控制信号进行交互。

从设计者的角度来看,寻找一种经济的解决方案去实现所需要的控制是很重要的。并行I/O端口提供了一套与外部输入和输出信号进行交互的便利方法。图9-1给出了微波炉的一种可能组成结构。它的关键之处在于使用的硬件总量非常小,一个简单的处理器带有一些够用的ROM和RAM部件,以及简单的用于连接该系统其他部分的输入输出接口。在单个的VLSI芯片中实现这些电路中的大部分功能是可行的。

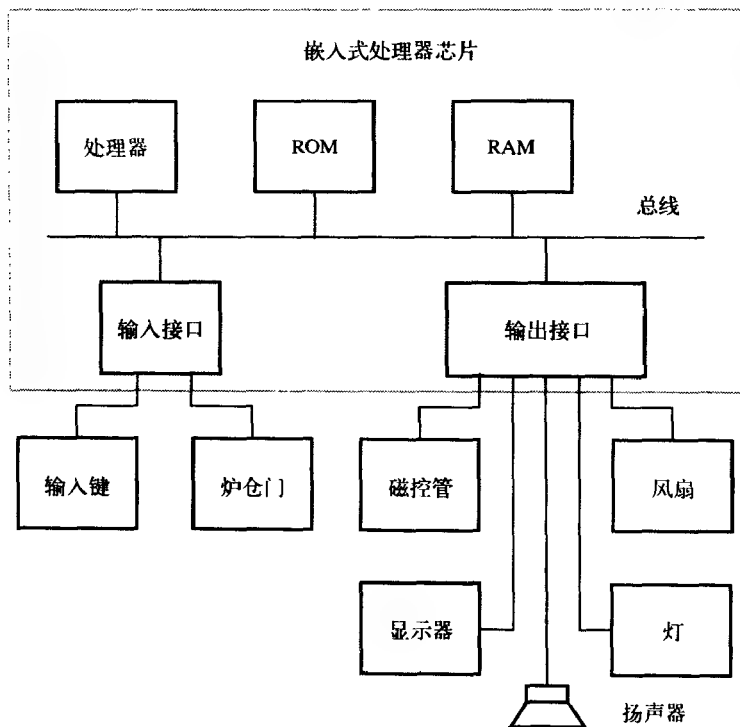


图9-1 一个微波炉的框图

### 9.1.2 数码照相机

数码照相机给出了一个在小型封装设备中成熟的嵌入式系统的优秀范例。图9-2列出了数码照相机中的主要部分。

传统照相机使用胶片去捕获图像。在数码照相机中，用一个光传感器阵列去捕获图形。这些传感器是基于发光二极管构造的，这些二极管可以将光转换成电荷。光的强度决定着产生电荷总量的大小。有两种不同类型的传感器用在商业产品中。一种类型是著名的电荷耦合器件 (CCD)。它是最早被用于数字照相机中的传感设备，并且它已经改进成可提供高质量图像的设备。最近基于CMOS技术的传感器已经开发出来，它们的价格较低，但是不能像CCD那样提供高质量的图像。

514

每个传感器产生一个对应于一个像素的电荷，它是图片图像中的一个点。像素的数量决定着可以记录和显示的图像的质量。电荷是一个模拟量，它被数模 (A/D) 转换电路转换成数字表示方式。A/D转换可以产生一个图像的数字表示形式，其中每个像素的颜色和色彩饱和度使用一个多位的数表示。图像的这种数字形式就可以使用标准的计算机电路进行操作了。

关键的功能部件是系统控制器。这个功能块包含着处理器、存储器 (RAM及EEPROM两种) 以及需要与系统其他部分连接的接口电路。处理器控制着照相机的操作，它处理原始图像数据，即从A/D电路中接收数据生成适合于计算机、打印机和显示设备表示的标准格式。主要使用的格式有：TIFF用于无压缩的图像，JPEG用于压缩的图像。

处理过的图像被存储在一个大型图像存储设备中，在5.3.5节中描述的闪存卡 (Flash memory cards) 就是一种用于图像存储的流行设备。其他的选择还有软盘和小型硬盘设备。

515

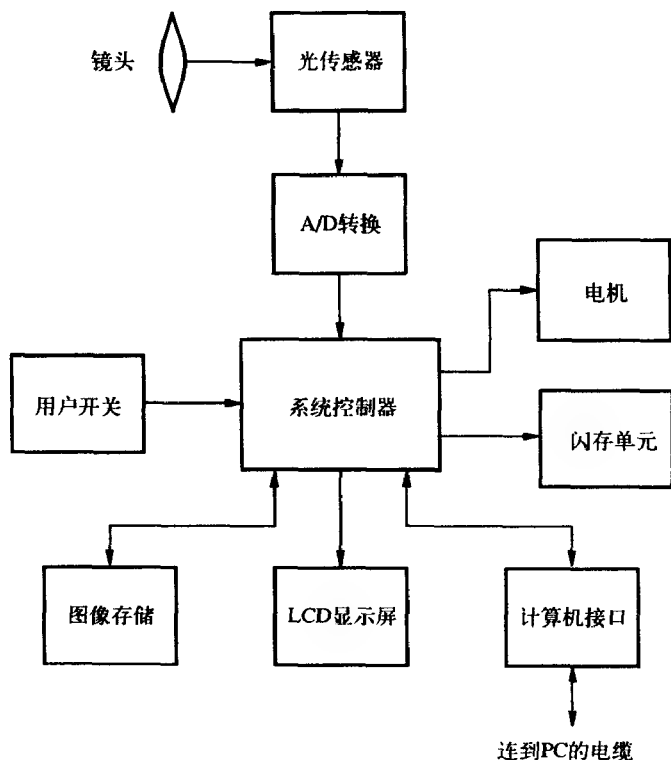


图9-2 一个简化的数码照相机框图

获取并处理过的图像可以显示在液晶显示器的屏幕上，该屏幕是嵌入在照相机中的。这里允许用户决定哪个图像值得保存，可以获取并保存的图像的数量与图像存储器的大小有关，同时也依赖于图像质量的选择，即每个图像中像素的个数。

标准接口提供将图像传递到计算机或打印机中的简单机制。这些接口可能是一个简单的串行或并行接口，或是一个针对标准总线（例如PCI或USB）的连接器。如果使用了闪存卡，图像也可以使用物理传递卡进行传递。

系统控制器还生成控制运动（用于对目标的聚焦）和闪光部件操作所需要的信号。有一些输入信息是由于用户对开关的操作而形成的。

数码照相机中所需要的处理器比以前描述的微波炉应用中的处理器要强大许多，该处理器必须执行相当复杂的信号处理功能。而且，重要的是该处理器不能消耗太多的能量，因为照相机是一种用电池做电源的设备。通常处理器消耗的电能耗比照相机中显示器和闪光灯所消耗的电能耗要少。

### 9.1.3 家用遥测技术

计算机在家庭中的应用正在快速增长，它们被当成通用的计算设备，也被用在大量的嵌入式应用中。在9.1.1节中我们考虑了微波炉的例子，在其他的设备中还可以找到类似的例子，像洗衣机、干燥机、洗碗机、炊具、火炉以及空调。另一个值得关注的例子是可视电话，电话中的这种嵌入式处理器使得电话可以具有多种有用的特征。除了标准的电话特征外，微处理器控制的电话可以提供对家庭中其他设备的远程访问功能，它也可以被当作计算机设备的通信设备。

来使用。

使用这种电话的人可以远程执行以下的功能：

- 与一个计算机控制的家居安全系统通信。
- 对火炉或空调保持的温度进行调整，设定一个合适的温度。
- 对提前已放入微波炉中的食品设定启动时间、烹调时间及加热温度。
- 读电、气和水表，取代过去服务公司派遣雇员到家中去读这些表的过程。

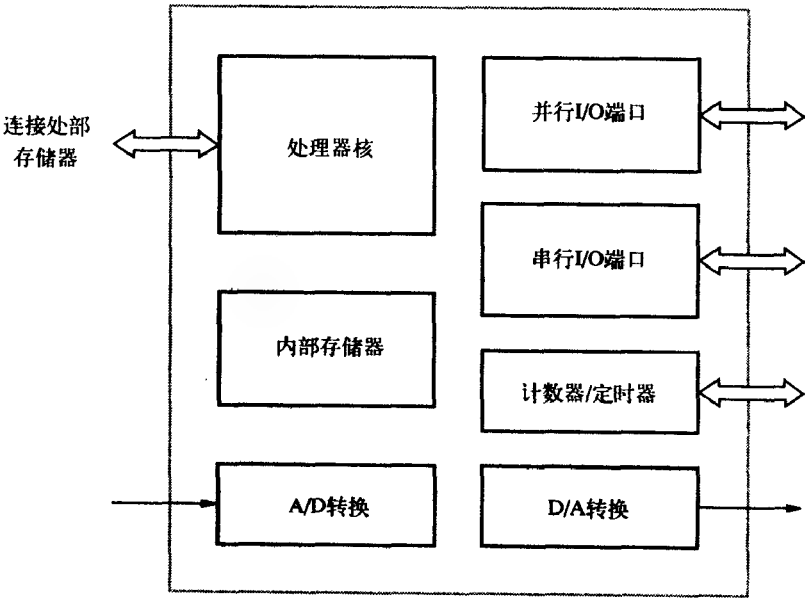
如果涉及的设备是由微处理器控制的，那么所有这些功能都很容易实现。因为只需要在设备中的微处理器与电话中的微处理器之间提供一条连接即可。这些连接可以用不同的方式实现。最简单的方式是使用按位串行通信，如果控制器芯片中包含有UART接口，这种通信很容易完成，这种类型的通信接口在4.6.2节中讨论过。用远距离的信号去观察并控制设备状态通常称为遥测技术。

516

9.2 嵌入式应用中的处理器芯片

包含有一个处理器、一些存储器和在嵌入式应用中用到的I/O接口电路的芯片，通常称作为嵌入式处理器。由于这样的芯片在应用中要执行重要的控制功能，并且它是基于微处理器实现的，因此它们也被称为微控制器芯片。

一个嵌入式处理器芯片应该能够满足为各种各样的应用提供服务的要求。图9-3给出了一个芯片的典型模块图。其中主要的部分是处理器核，它是商用微处理器的一个基础。选择在实际中已经证明被大众喜爱的微处理器体系结构是较稳妥的，因为对这样的处理器已经有许多CAD工具、优秀的书籍以及大量有助于新产品设计的经验和知识。



517

图9-3 嵌入式处理器框图

在芯片中包含一些存储器是非常有用的，你可能会发现在小型应用中这些存储器可能完全能够满足应用需求中的存储要求。这些存储器中有些必须是RAM类型的，用于存储那些在计算

中发生变化的数据；有些应该是ROM类型的，用来存储软件，因为嵌入式系统中通常不包括磁盘驱动器。为了满足低容量应用中的成本效益，需要有一种可现场编程的ROM类型，而实现这种存储的流行选择是EEPROM和闪存。

有几种I/O端口可以提供并行和串行的接口。这些接口可以容易地实现标准的I/O连接。在许多的应用中，需要以可编程时间间隔生成控制信号。如果嵌入式处理器芯片中包含一个定时器电路，这类任务就很容易完成。因为定时器是一个时钟脉冲计数电路，所以它也可以用作计数，比如对一个给定的输入线的脉冲个数进行计数。

在一个嵌入式系统中可能包含一些模拟设备，为了能与这样的设备进行交互，就需要能够将模拟信号转换成数字表示形式，并且反之亦然。如果嵌入式控制器中包含有A/D和D/A转换电路，这些也是很容易做到的。

许多嵌入式处理器芯片在市面上可以得到，一些比较好的芯片是：Motorola的68HC11、683xx 及MCF5xxx系列，Intel的8051和 MCS-96 系列，它们使用CISC型处理器核，而ARM微控制器具有RISC型的处理器。处理器核的特性在本章不是重要的内容，我们强调的是嵌入式应用系统方面的内容，并说明如何将前几章的概念结合起来用到一个完整的嵌入式计算机系统的设计中去。

### 9.3 一个简单的微控制器

在这一节中我们讨论一个简单微控制器的基本构成，说明在实际中有哪些典型特征会被用到。图9-4给出了它的框图，其中包含一个处理器核和一些片上存储器。因为片上存储器可能满足不了所有潜在的应用，因此在该芯片的管脚上提供了处理器的总线连接，以便可以添加外部存储器。

它有两个8位并行接口A和B和一个串行接口。该微控制器还包含一个32位的计数器/定时器电路，该电路可以用于以编程时间间隔产生内部中断、作为系统的时钟、对输入线上的脉冲个数计数、生成各种循环的方波输出信号等。

#### 9.3.1 并行I/O端口

并行接口提供的I/O能力类似于在图4-34中描述的方案。在A和B端口上每个独立的端口连线都可以被用作输入或是输出，输入/输出的方向是由存储在数据定向寄存器中的位模式决定的。图9-5说明了对端口A上的某一位的双向控制。如果数据定向触发器的值为0时，端口引脚PA<sub>i</sub>就被当作一个输入。在这种情况下，一个激活的控制信号Read\_Port的逻辑值被放在处理器总线的数据线D<sub>i</sub>端口上。在这个输入线路中不包含存储单元（相当于图4-34中的DATAIN），因此处理器从端口管脚上直接读该数据。如果数据定向触发器中设置的值为1，端口管脚作为输出使用。这种情况下，在控制信号Write\_Port的控制下，加载到数据输出触发器上的逻辑值被放置在引脚处。由于数据的方向位是按每一个引脚提供的，因此一些引脚可以被编程为输入，而另一些可以作为输出。

端口A和B之间的数据传递操作涉及到的八个8位寄存器，在图9-6中给予了描述。图中还给出了这些寄存器的存储器映射地址。这里我们任意选择了32位地址范围中的高端地址。

状态寄存器PSTAT包含着状态标志。当端口A的管脚上有一个新到的数据时，PASIN标志被置成1。当处理器用读PAIN寄存器的操作接收了该数据后，它又被清成0。当寄存器PASOUT中

的数据被传递到连接设备时，PASOUT标志置成1，表示处理器现在可以向PAOUT中装入新数据了（控制线上标志传递到设备的信号在下文中描述）。当处理器将数据写入PAOUT时，PASOUT标志被清成0。PBSIN和PBSOUT标志对端口B执行同样的功能。

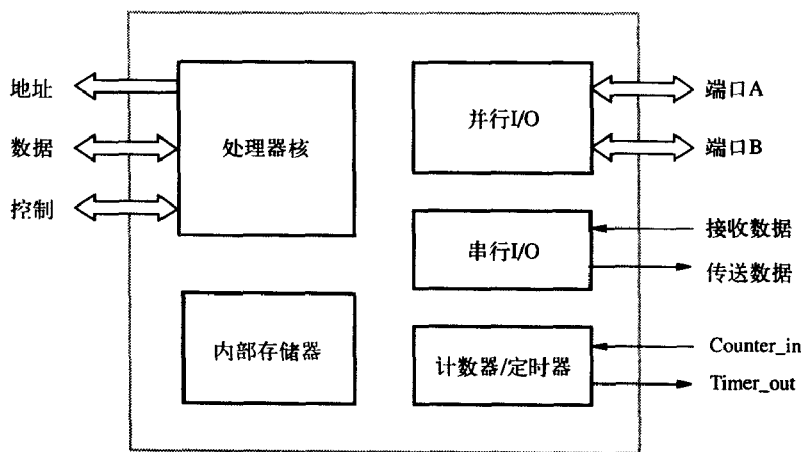


图9-4 一个微控制器的例子

状态寄存器还包含了四个中断标志。当某个中断被允许并且相应的I/O动作发生时，就有一个中断标志（比如IAIN）被设置成1。中断允许位保存在控制寄存器的PCONT中。PCONT中的某个允许位设置成1就可以允许相应的中断发生。例如，如果ENAIN=1并且PASIN=1，则中断标志IAIN被置成1并且产生一个中断请求。因此，

$$IAIN = ENAIN \cdot PASIN$$

就有一个单独的中断请求信号被使用。作为对某个中断请求的响应，处理器必须检查该中断标志以确定该中断请求的实际来源。

状态和控制寄存器中的信息用于控制传递到或来自于连接到端口A和端口B上设备中的数据。端口A有两条控制线CAIN和CAOUT，它们可以用于为接口和连接设备间提供以下的信号机制：当设备将一个新数据放置在端口引脚时，它表示在一个时钟周期中该动作被一个CAIN连线激活。当接口电路观测到CAIN=1时，它将状态位PASIN置成1。稍后，当处理器读出了这个输入数据时该位被清成0。这个动作还引发该接口向CAOUT连线发送一个脉冲，通知该设备可以向接口发送新的数据了。对于输出传递，处理器将数据写入PAOUT寄存器中，这个动作也将PASOUT位清成0并向CAOUT连线发送一个脉冲，通知设备可以得到一个新数据了。当设备拿到该数据时，它再向CAIN连线发送一个脉冲指明这个动作，它又将PASOUT变换成1。当一个端口上的所有数据引脚有着同样的方向时，也就是当端口都作为输入或输出时，这种信号机制是可使用的。如果有些引脚选择的是输入而有些选择的是输出，

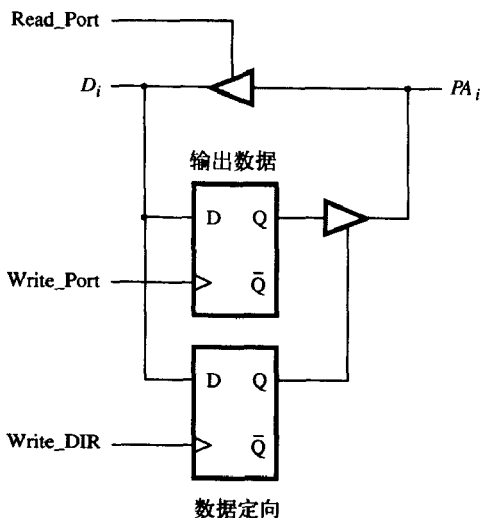


图9-5 访问图9-4端口A中的一位

此时控制线以及状态和控制寄存器中均不包含有用的信息。

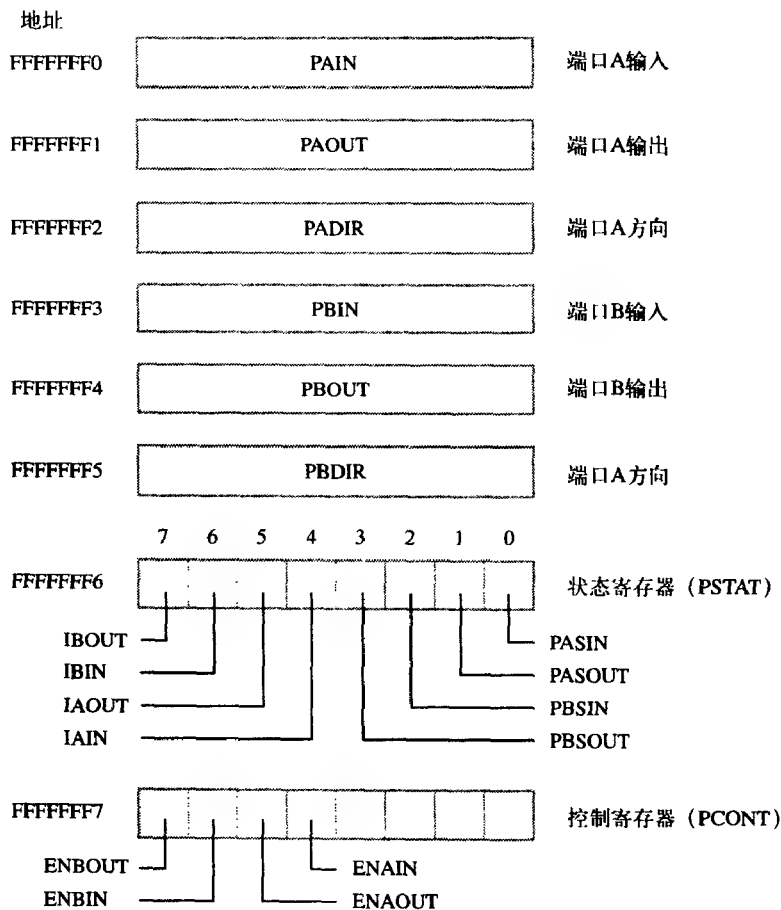


图9-6 并行接口寄存器

### 9.3.2 串行I/O接口

串行接口基于在图4-37中说明的原理提供UART (Universal Asynchronous Receiver Transmitter, 通用异步收发器) 功能的数据传递。在发送与接收路径中使用了双缓冲, 如图9-7所示。这种缓冲是为了解决I/O传递中的瞬间冲突。

图9-8给出串行接口的可编址地址寄存器, 输入数据是从8位接收缓冲区中读出的, 输出数据是送入到8位传送缓冲区中去的。状态寄存器SSTAT提供有关接收和传送单元的当前状态信息。当在接收缓冲区中存在有效的数据时, SSTAT<sub>0</sub>位被置成1。当传送缓冲区为空并可以装入新数据时, SSTAT<sub>1</sub>被置成1。这些位与在4.1节中描述的状态特征位SIN和SOUT起着相同的作用。如果在接收过程中产生了一个错误, SSTAT<sub>2</sub>就被置成1。例如, 如果接收缓冲区中的数据在还未被处理器读出之前被后面接收的字符覆盖, 就产生一个错误。状态寄存器中还包含有中断标志, 当接收缓冲区已满并且接收中断是允许的, SSTAT<sub>4</sub>位被置成1。类似地, 当传送缓冲区为空并且传送中断是允许的, SSTAT<sub>5</sub>位被置成1。如果SSTAT<sub>4</sub>或是SSTAT<sub>5</sub>等于1时, 串行接口就引发一个

中断。如果 $SSTAT_6=1$ 或者 $SSTAT_2=1$ ，并且错误条件中断是允许的，它也引发一个中断。

控制寄存器SCONT用来保存中断允许位。设定 $SCONT_{6-4}$ 位为1或0，分别表示允许或禁止相应的中断发生。该寄存器中还说明如何生成传送时钟，如果 $SCONT_0=0$ ，则传送的时钟与系统（处理器）时钟相同；如果 $SCONT_0=1$ ，则用时钟除法电路获得传送时钟。

串行接口中的最后一个寄存器是时钟除数寄存器DIV。这个32位的寄存器与计数器电路相关联，用它除以系统时钟信号可以生成串行传送时钟。计数器生成一个时钟信号，它的频率等于系统时钟被该寄存器中的内容除所得到的频率。存入该寄存器的这个值被传递到计数器中，然后系统时钟进行递减计数。当计数减为零时，该计数器用DIV寄存器中的值重新装入。

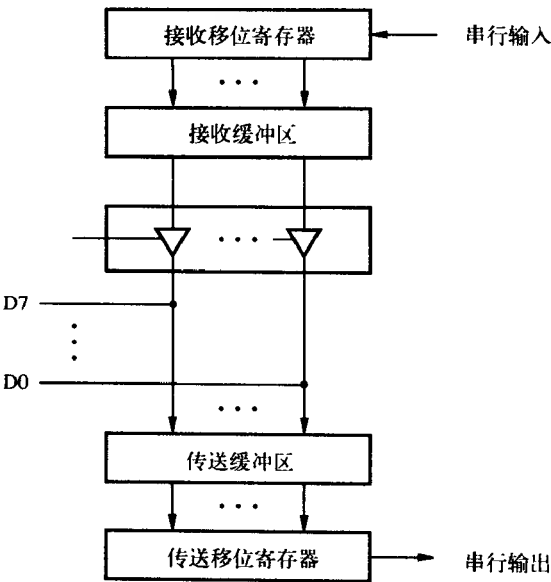


图9-7 串行接口中的接收和传送结构

522

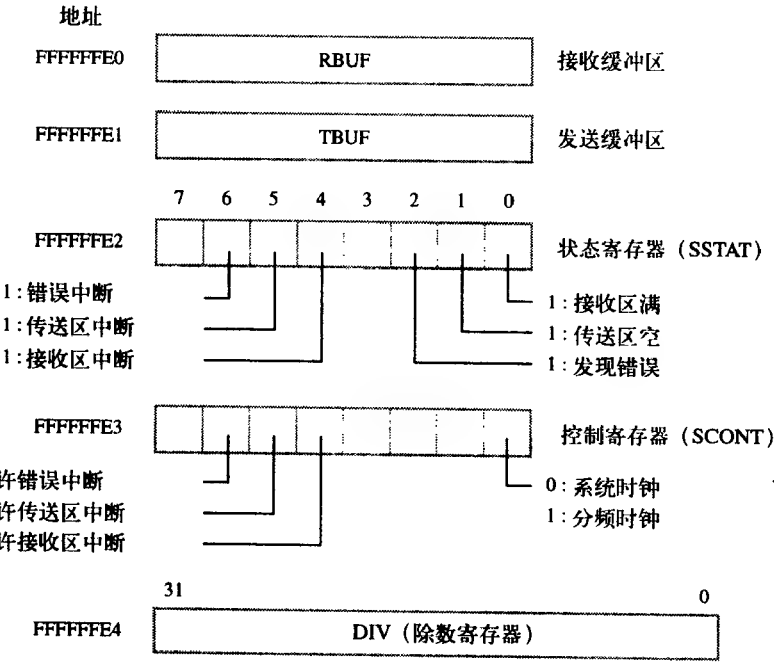


图9-8 串行接口寄存器

9.3.3 计数器/定时器

32位的递减计数器电路可以作为计数器或定时器使用。该电路的基本操作包括将一个初始值加载到计数器中，然后使用内部系统时钟或是外部时钟信号递减计数器中的内容。这个电路可用编程实现，当计数器中的内容达到0时引发一个中断。图9-9给出了与计数器/定时器电路相



连的寄存器。计数器/定时器的寄存器CNTM可以被装入一个初值，然后将它传递到计数器电路中。计数器中的当前内容可以用访问内存地址FFFFFFD4的方式读出。控制寄存器CTCON是用于指明计数器/定时器电路的操作方式的。它提供了一种用于开始和停止计数处理以及当计数器内容递减为0时允许中断的机制。状态寄存器CTSTAT反映该电路的状态。

523

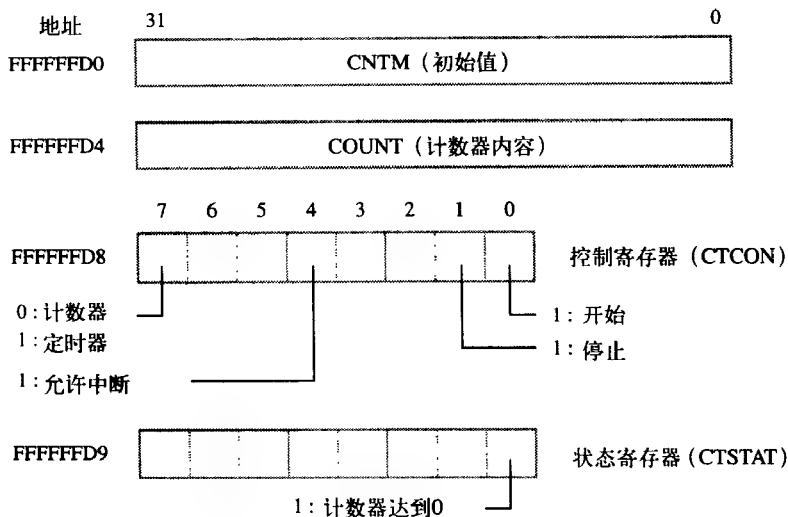


图9-9 计数器/定时器寄存器

### 计数器模式

设定CTCON<sub>7</sub>为0，可选择该电路为计数器模式，初始值用写入寄存器CNTM的方式写入计数器中。当CTCON<sub>0</sub>位被程序指令设置成1时计数过程开始。一旦计数器开始计数，CTCON<sub>0</sub>位自动清为0。计数器被计数器输入 (Counter\_in) 线上的脉冲递减。当达到0时，计数器电路将状态标志CTSTAT<sub>0</sub>被置成1，并且如果相应的中断允许位已经被置成1，将引发一个中断。在下一个时钟脉冲到来时计数器重新装入一个初值，这个初始值是被保存在寄存器CNTM中的，这时计数过程继续。当CTCON<sub>1</sub>被设置为1时计数过程停止。

### 定时器模式

524 设定CTCON<sub>7</sub>为1时选择该电路为定时器模式，这种模式适于在图9-4中的输出线Timer\_out上生成一个方波信号。这个过程开始如同上面对计数器模式的解释一样，当计数器进行递减计数时，输出线上的值保持着一个常数。直到减到0时，计数器自动用初始值进行重新加载，这时连线上的输出信号反向。因此，输出信号的周期是控制时钟脉冲起始计数值的两倍。在定时器模式下，计数器用系统时钟作递减操作。

### 9.3.4 中断控制机制

微控制器有两个中断请求线IRQ和XRQ。IRQ线用于由微控制器内部I/O接口引发的中断；XRQ线用于由外部设备引发的中断。当处理器发现IRQ线被激活时，它用轮询方式确定该中断请求源。完成这个过程就是对状态寄存器的标志位PSTAT、SSTAT及CTSTAT进行轮询。XRQ中断的优先级比IRQ中断的优先级高。

处理器中的状态寄存器PSR有两个允许中断位。如果PSR<sub>6</sub>=1，IRQ中断被允许；如果

PSR<sub>7</sub>=1, XRQ中断被允许。当处理器接收一个中断时,它将在中断服务程序执行前清除相应的PSR位,禁止同一优先级上的其他中断产生。这里要用到一个中断向量表,其中对应于IRQ和XRQ的中断向量分别存储在单元\$24和\$28中。每个中断向量中包含着相应的中断服务程序的第一条指令的地址。

在处理器中有一个链接寄存器LR,它如同在2.9节中讲述的那样,被用做子程序的链接。当有一个子程序调用指令对程序计数器PC的内容进行更新时,PC中需要返回的地址被保存在LR中,然后用预定好的转移子程序的第一条指令更新。当接收一个中断请求时,将发生同样的动作。但这时,除了将返回地址保存在LR中以外,处理器状态寄存器PSR中的内容还应保存在处理器的寄存器IPSR中。

从子程序返回是执行一个ReturnS指令完成的,该指令将LR中的内容传递到PC中。从中断中返回是执行ReturnI指令完成的,该指令分别将LR和IPSR中的内容传递到PC和PSR中。因为只有一个LR和IPSR寄存器,嵌套式中断可以用在中断服务程序中使用一条指令,将这些寄存器中的内容存储在堆栈中的方法来实现。这种策略类似于4.3.1节中讨论的ARM解决方案。

## 9.4 程序设计问题

前面已经介绍了微控制器的硬件,现在我们来考虑一些软件方面的问题。可以用汇编语言或高级语言来编写程序。后一种选择在大部分应用中更可取,因为这样所需要的编码容易生成并且容易维护,而且开发时间缩短。我们将用两种语言方式给出一些实例。本节中的例子是一些最基本的应用,我们的主要目的是为了说明实现方法的可能性。在9.5节中,我们将给出一个完整且比较复杂的实例,并且选择C语言作为高级编程语言。

525

考虑以下任务,微控制器要从一个串行位的数据源处将8位字符传递到并行位。数据源连接到串行接口上,目标被连接到并行端口A上。当接收缓冲区中存有一个字符时,通过将状态寄存器的相应位置成1来表示,也就是SSTAT<sub>0</sub>=1。并行端已经被设置成了输出方式,这是通过将数据定向寄存器中所有位都置成1(PADIR<sub>7-0</sub>=\$FF)完成的。我们假定输出设备通过端口A接收字符的速度比通过串行口提交字符的源设备要快,这样就不需要去轮询端口A,看它是否准备好接收下一个字符了。首先我们用轮询方式说明如何完成所需要的传递,然后再用中断方式来实现同样的任务。

### 9.4.1 轮询方法

如同在2.7节中描述的那样,轮询方式需要反复地测试状态标志,直到接收到一个字符为止。在我们的例子中,需要轮询的是SSTAT<sub>0</sub>位。

#### 汇编语言程序

图9-10说明了使用汇编语言如何实现这一任务。我们用在第2章中使用过的一般格式,对微控制器中的I/O寄存器使用符号名与寄存器的地址相结合的方式说明。程序不断地循环检查状态位SSTAT<sub>0</sub>。一旦SSTAT<sub>0</sub>=1,接收缓冲区RBUF中的字符就被传递到了端口A上。调用执行对RBUF的读操作自动清除SSTAT<sub>0</sub>位。如果PSTAT<sub>1</sub>=1,该字符被写到PAOUT寄存器中。

图9-10中的程序使用一个无限循环的方法传递一个连续的字符流。在实际应用中此处不应该使用无限循环,因为如果这样其他的任务可能也会因此变得复杂起来。我们此处使用无限循环只是为了使例子简单化。

RBUF	EQU	\$FFFFFFE0	接收缓冲区
SSTAT	EQU	\$FFFFFFE2	用于串行接口的状态寄存器
PAOUT	EQU	\$FFFFFFF1	端口A的输出数据
PADIR	EQU	\$FFFFFFF2	端口A的定向寄存器
* 初始化			
	ORIGIN	\$1000	
	MoveByte	#\$FF,PADIR	配置端口A为输出端口
* 传送字符			
LOOP	Testbit	#0,SSTAT	查看是否有准备好的新字符
	Branch=0	LOOP	
	MoveByte	RBUF,PAOUT	传送一个字符到端口A
	Branch	LOOP	

图9-10 使用轮询方式完成字符传送的一个通用汇编语言程序

### C程序

在C语言程序中，存储器映射的I/O单元可以使用指针变量来表示，指针变量中的值是这个单元的地址。如果这个位置的内容被当成字符看待，该指针应该说明成字符类型。这样定义该内容的长度为一个字节，它是I/O寄存器的大小。寄存器的内容可以方便地转换成16进制形式。

图9-11给出了一个实现上述任务的C语言程序。Define语句用来将需要的地址与指针的符号名关联起来。这些语句与图9-10中的EQU语句的作用相同。它们有助于C编译器的预处理程序用实际值去替换程序中的符号名。编译后的代码类似于图9-10中的代码。

注意这里RBUF和SSTAT指针被声明成了可变的（volatile）。这样做是因为这个程序只能读相应地址中的内容，既不能向这些地址写任何数据，也不能将这些地址与特定的值相关联。优化编译程序可能会删除那些看来不会产生影响的语句，这些语句中包含一些有关的变量，但它们的值从未被改变过。因为RBUF和SSTAT寄存器的内容改变是受程序外部因素影响的，所以非常有必要将这个事实告诉编译程序。编译程序不能删除那些包含可变变量的语句。

图9-12举例说明了另一种不同的方法。我们不将图9-11中指向I/O寄存器的指针定义成常数，而将指针声明成变量，它指向保存字符类型数据的单元。这样，像RBUF和PAOUT这样的符号就代表存放着I/O寄存器实际地址的内存单元的地址。在这种情况下，编译后的代码就可能表现为如图9-13中所示的内容。当要访问一个指定的I/O寄存器时，它的地址被送入处理器的一个寄存器中，然后使用寄存器间接寻址方式去访问所需的I/O寄存器。注意，图中给出的仅仅是编译后代码中的一部分内容，编译程序也给出了与符号RBUF、SSTAT、PAOUT以及PADIR相关的内存地址值。从该图程序中产生的机器码会比从图9-11中产生的机器码多。在以后的例子中我们将使用图9-11中使用的方式来定义指针。使用define语句指定I/O单元的地址所强调的事实是：这些地址是常数，它们在程序执行中从不发生变化。

在本章的C程序中，可以包括一个特定的值，它在执行给定动作的语句时被直接地存入到了寄存器中。例如，在图9-11中有语句：

```
*PADIR = 0xFF
```

该语句将寄存器PADIR的8位都置成了1，使得端口A成为一个输出端。在C程序编写中更常用的方法是为这样的常数值声明一个名字，然后在以后的程序中使用这个名字。这样选择的目的是

是为了保证数字尽可能地小，并且使得它容易与I/O寄存器的具体值进行比较，如图9-6到图9-9所示。

```

/* 定义寄存器地址 */
#define RBUF (volatile char *) 0xFFFFFE0
#define SSTAT (volatile char *) 0xFFFFFE2
#define PAOUT (char *) 0xFFFFF1
#define PADIR (char *) 0xFFFFF2

void main()
{
    /* 初始化并行端口 */
    *PADIR = 0xFF;          /* 将端口A配置为输出端口 */

    /* 传送字符 */
    while (1) {              /* 无限循环 */
        while ((*SSTAT & 0x1) == 0); /* 等待一个新字符 */
        *PAOUT = *RBUF;      /* 将字符传输到端口A */
    }
}

```

527

图9-11 使用轮询方式传送字符的C程序

```

/* 定义寄存器地址 */
volatile char *RBUF = (char *) 0xFFFFFE0;
volatile char *SSTAT = (char *) 0xFFFFFE2;
char *PAOUT = (char *) 0xFFFFF1;
char *PADIR = (char *) 0xFFFFF2;

void main()
{
    /* 初始化并行端口 */
    *PADIR = 0xFF;          /* 将端口A配置为输出端口 */

    /* 传送字符 */
    while (1) {              /* 无限循环 */
        while ((*SSTAT & 0x1) == 0); /* 等待一个新字符 */
        *PAOUT = *RBUF;      /* 将字符传送到端口A */
    }
}

```

图9-12 另一种使用轮询方式传送字符的C程序

528

	Move	PADIR,R0
	MoveByte	#FF,(R0)
LOOP	Move	SSTAT,R0
	Testbit	#0,(R0)
	Branch=0	LOOP
	Move	RBUF,R0
	Move	PAOUT,R1
	Move	(R0),(R1)
	Branch	LOOP

图9-13 图9-12中的程序段编译后可能生成的代码

### 9.4.2 中断方法

作为替换SSTAT<sub>0</sub>位循环轮询来检测新字符的方法，我们可以将I/O接口配置成当SSTAT<sub>0</sub>=1时引发一个中断请求。这时相应的SCONT寄存器中的中断允许位SCONT<sub>4</sub>必须被置成1，还需要在处理器中将PSR<sub>6</sub>置成1使得IRQ中断被允许。这些可以通过将\$40装入到PSR来实现，这样也禁止了XRQ中断。中断服务程序的地址必须存放在存储单元\$24中。

#### 汇编语言程序

图9-14 给出了一个汇编程序，它以中断方式实现字符传送的例子。当SSTAT<sub>0</sub>=1时，引发一个中断请求。在中断应答中，中断服务程序从RBUF中将一个字符传递到PAOUT中。在读接收缓冲区RBUF内容的同时还要将SSTAT<sub>0</sub>位清成0。注意，这里程序中没有引用SSTAT状态寄存器中的内容，与以前的范例一样此处再次采用了一个无限循环语句等待一个新字符的产生，这样可以保持例子的简单化。

#### C程序

使用中断方式写C程序，我们需要解决两个问题：

- 如何访问处理器的寄存器？
- 如何写中断服务程序？

中断实现方式要求处理器状态寄存器中的中断控制位做适当的设置。这就涉及对PSR的写操作。在高级语言（比如C）中，变量在编译代码中用存储单元来表示。因此，存储器映射I/O寄存器可以通过直接的方式进行处理，就像我们在图9-11和图9-12中做的那样。但是，处理器寄存器（比如状态寄存器PSR）没有相应的内存地址与其对应，这些寄存器在C程序中可以用直接包含相应的汇编语言指令的方式进行访问。例如以下语句：

```
__asm__( "Move #0x40,%PSR" )
```

使得C编译程序将以下汇编指令

```
Move #$40, PSR
```

**529** 插入到编译后的代码中，这样就可以将模式\$40装入到PSR寄存器中了。

RBUF	EQU	\$FFFFFFE0	接收缓冲区
SCONT	EQU	\$FFFFFFE3	串行接口的控制寄存器
PAOUT	EQU	\$FFFFFFF1	端口A输出数据
PADIR	EQU	\$FFFFFFF2	端口A的定向寄存器
* 初始化			
	ORIGIN	\$1000	
	MoveByte	#\$FF,PADIR	配置端口A作为输出端口
	Move	#INTSERV,\$24	设置中断向量
	Move	#\$40,PSR	处理器响应IRQ中断
	MoveByte	#\$10,SCONT	允许接收端中断
* 传送循环			
LOOP	Branch	LOOP	无限等待循环
* 中断服务程序			
INTSERV	MoveByte	RBUF,PAOUT	传送一个字符到端口A
	Returnl		从中断返回

图9-14 使用中断方式完成字符传送的通用汇编语言程序

第二个问题是中断服务程序，这种程序必须写成C程序中的一个函数。而编译程序处理一个函数就像处理子程序的过程一样。图9-15给出了一个例子，其中有一个主程序，它执行一些任务，这里没有写出该程序的任何语句。还有一个叫做intserv的函数，它仅仅完成从接收缓冲区RBUF中向输出端口PAOUT传送一个字符的动作。编译程序为函数intserv生成以下指令代码：

```
Move $FFFFFFE0, $FFFFFFF1
```

```
ReturnS
```

现在看看如果将intserv看作一个中断服务程序会发生什么情况。在使用中断时，从中断服务程序中返回必须使用中断返回指令ReturnI。该指令使得程序计数器和处理器状态寄存器恢复它们原有的值。因此，我们必须在程序中用以下语句插入ReturnI指令：

```
__asm__( "ReturnI" )
```

所以编译后的代码应该是：

```
Move $FFFFFFE0,$FFFFFFF1
```

```
ReturnI
```

```
ReturnS
```

显然，在这种情况下ReturnS指令将永远不会被执行。

现在可以写使用中断方式所需要的程序了。图9-16给出了一个沿用图9-11风格的中断处理方式的程序。注意，指向I/O寄存器的指针是字符型，因为它是指向存放着1字节数据的单元。然而指针int\_addr是无符号整型的，因为它指向一个存储着4字节中断向量的内存单元。

## 9.5 I/O设备的时序限制

在以上几节的举例中由于假定连接到端口A上的输出设备比通过串行口提供字符的设备要快，从而使问题简化了。在实际系统中涉及到的设备会具有各种不同的速度。假定我们将以前举例中的速度做个反向调整，即让输出设备比输入设备慢。这就意味着字符不能够从RBUF到PAOUT直接进行发送。而是需要将它们临时存储在内存缓冲区中。这种缓冲区必须按先进先出（first-in-first-out，FIFO）队列组成。简单FIFO的问题是指向队列的头和尾指针被不断地进行增值，从而使得该队列随着字符的存储和检索在整个存储区中移动。一个较好的解决方法是使用一个环形缓冲区，也就是一个环形队列，它由固定数量的内存单元构成并且当到达缓冲区末尾时返回。当然，如果在一个脉冲串里快速的源设备生成的字符远比在该脉冲串里输出设备所能接收的字符多得多，这个缓冲区就会溢出。为了避免出现这种溢出状况，我们假定源设备在一个脉冲串中生成的字符少于80个，而输出设备在下一个脉冲串到达之前可以接收这些字符。这就意味着该环形缓冲区必须能够保存80个字符。

```
#define RBU (volatile char *) 0xFFFFFEE0
#define PAOUT (char *) 0xFFFFFFF1

.
.
.

void main()
{
.
.
.
}

void intserv()
{
    *PAOUT = *RBU; /* 传送一个字符到端口A */
}
```

图9-15 C程序中的一个函数调用

530

531

```

/* 定义寄存器地址 */
#define RBUF (volatile char *) 0xFFFFFEE0
#define SCONT (char *) 0xFFFFFEE3
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define int_addr (int *) (0x24)

void intserv();

void main()
{
    /* 初始化并行端口 */
    *PADIR = 0xFF; /* 配置端口A作为输出端口 */

    /* 初始化中断机制 */
    int_addr = &intserv; /* 设置中断向量 */
    __asm__("Move #0x40,%PSR"); /* 处理器响应IRQ中断 */
    *SCONT = 0x10; /* 允许接收端中断 */

    /* 传输字符 */
    while (1); /* 无限循环 */
}

/* 中断服务程序 */
void intserv()
{
    *PAOUT = *RBUF; /* 传送字符到端口A */
    __asm__("ReturnI"); /* 从中断返回 */
}

```

图9-16 使用中断方式用于字符传送的C程序

我们可以用一个8位的数组，用两个变址值表示该队列头和尾的当前位置来实现环形缓冲区。当这些变址值相等时，表示该队列为空。

### 9.5.1 通过环形缓冲区做传送的C程序

图9-17给出了一个使用环形缓冲区传送字符的C程序的例子。缓冲区是一个叫做 mbuffer的数组，它有80个数据单元。字符使用变址fin存入缓冲区，它们通过使用变址fout被检索。每做一次循环，首先测试状态寄存器SSTAT，查看在RBUF中是否有一个新的字符存在，因为从较快的设备中传送字符时必须给予较高的优先级。如果RBUF中没有字符，同时环形缓冲区非空并且端口准备就绪，就向端口A执行一次传送。每做一次传送其变址值就进行一次更新。

### 9.5.2 通过环形缓冲区做传送的汇编语言程序

图9-18给出了一个用汇编语言实现的例子。环形缓冲区使用变址寻址方式进行访问。寄存器R0指向队列的第一个单元，寄存器R1和R2分别是队列的首尾变址。以下程序中的主要内容与图9-17中的程序相同。

## 9.6 反应计时器实例

前面已经介绍了微控制器的基本特征，现在说明如何将它用在一个简单的应用中。典型的嵌入式应用例子将会涉及相当复杂的内容，所以我们选择了一个简单并且容易理解的任务。

```

/* 定义寄存器地址 */
#define RBUF (volatile char *) 0xFFFFFEE0
#define SSTAT (volatile char *) 0xFFFFFE2
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PSTAT (volatile char *) 0xFFFFFFF6
#define BSIZE 80

void main()
{
    unsigned char mbuffer[BSIZE];
    unsigned char fin, fout;
    unsigned char temp;

    /* 初始化端口A和环形缓冲区 */
    *PADIR = 0xFF; /* 配置端口A作为输出端口 */
    fin = 0;
    fout = 0;

    /* 传送字符 */
    while (1) { /* 无限循环 */
        while ((*SSTAT & 0x1) == 0) { /* 等待新字符 */
            if (fin != fout) { /* 如果环形缓冲区非空 */
                if (*PSTAT & 0x2) { /* 输出设备准备就绪 */
                    *PAOUT = mbuffer[fout]; /* 发送一个字符到端口A */
                    if (fout < BSIZE - 1) /* 更新输出变址 */
                        fout++;
                    else
                        fout = 0;
                }
            }
        }
        mbuffer[fin] = *RBUF; /* 从接收缓冲区读取一个字符 */
        if (fin < BSIZE - 1) /* 更新输入变址 */
            fin++;
        else
            fin = 0;
    }
}

```

图9-17 通过环形缓冲区做传送的C程序

我们需要设计一个“反应计时器”，它可以用于测量人们对于视觉刺激的反应速度。设计思想是需要有一个微控制器去打开一盏灯，然后测量反应时间，即被测者按下开关关上灯的时间。详细的系统描述如下：

- 有两个手动按钮开关Go和Stop,一个发光二极管(LED)和一个三位数字的七段显示器。
- 按下Go开关,该系统被激活。
- 激活后,七段显示被设置成000,并且LED是关闭的。
- 经过三秒的延时,LED被打开并且计时过程开始。
- 当Stop开关按下时,计时过程停止,LED关闭,并且将经历的时间显示在七段显示器上。
- 经历的时间被计算出来,并按百分之一秒格式显示。由于显示中只有三位数,所以假定经历的时间应该是少于10秒的。



RBUF	EQU	\$FFFFFFE0	接收缓冲区
SSTAT	EQU	\$FFFFFFE2	串行接口的状态寄存器
PAOUT	EQU	\$FFFFFFF1	端口A输出数据
PADIR	EQU	\$FFFFFFF2	端口A的定向寄存器
PSTAT	EQU	\$FFFFFFF6	并行接口的状态寄存器
MBUFFER	ReserveByte	80	定义环形缓冲区
* 初始化			
	ORIGIN	\$1000	
	MoveByte	#\$FF,PADIR	配置端口A为输出端口
	Move	#MBUFFER,R0	R0指向缓冲区
	Move	#0,R1	初始化头指针
	Move	#0,R2	初始化尾指针
* 传送字符			
LOOP	Testbit	#0,SSTAT	检查是否有新字符准备就绪
	Branch $\neq 0$	READ	
	Compare	R1,R2	检查队列是否为空
	Branch=0	LOOP	队列为空
	Testbit	#1,PSTAT	检查端口A是否就绪
	Branch=0	LOOP	
	MoveByte	(R0,R2),P AOUT	发送一个字符到端口A
	Add	#1,R2	增加尾指针
	Compare	#80,R2	指针是否超出队列的界限
	Branch<0	LOOP	
	Move	#0,R2	循环回来
	Branch	LOOP	
	MoveByte	RBUF,(R0,R1)	将新字符放入队列
	Add	#1,R1	增加头指针
READ	Compare	#80,R1	指针是否超出队列的界限
	Branch<0	LOOP	
	Move	#0,R1	循环回来
	Branch	LOOP	

图9-18 通过环形缓冲区进行传送的通用汇编语言程序

图9-19描绘了实现反应计时器所需功能的硬件。这里微控制器提供了除输入开关和输出显示以外的所有功能。

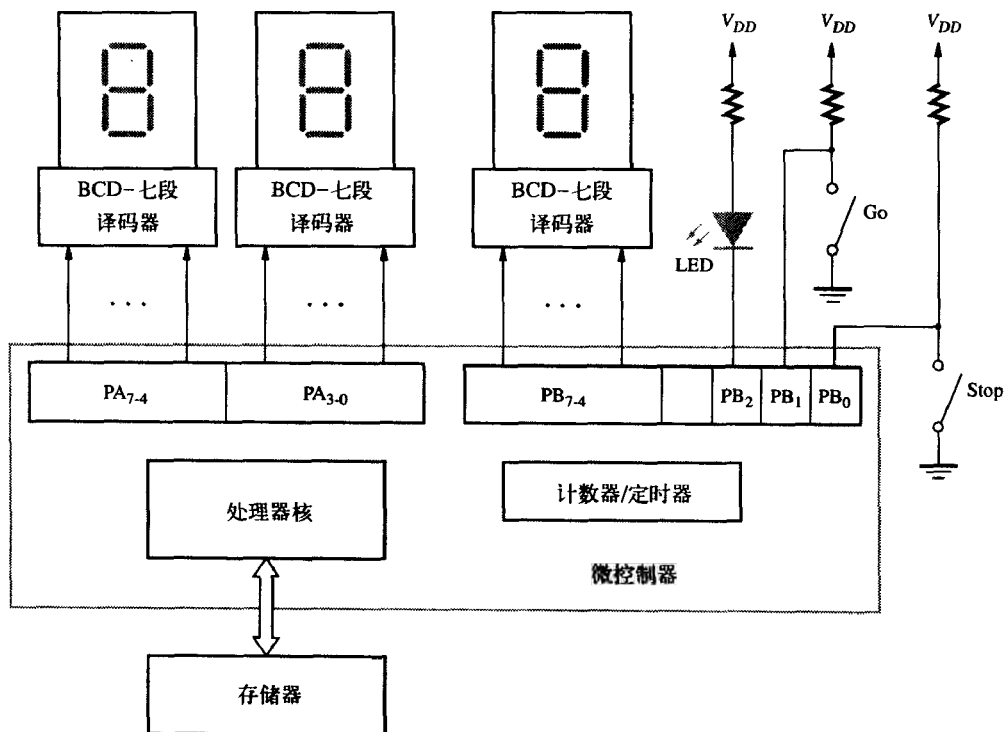
我们使用并行端口A和端口B完成所有的输入/输出功能。显示时间的前两个有效BCD位连接到端口A上，最后一个有效位连接到端口B的高四位上。如图所示，开关和LED连接到了端口B的低四位上。计数器/定时器电路用于测量经历的时间。它由系统时钟驱动，假定系统时钟具有100MHz的频率。

实现要求的程序可以基于以下方法完成：

- 测试用户的反应可利用一个等待循环监视程序完成，在循环中反复查询Go开关的状态。
- 当观察到Go开关已关闭，也就是已发现 $PB_1 = 0$ 时，再延时三秒钟之后打开LED。
- 计数器设置成初始值\$FFFFFFF，并且计数过程被激活；每个时钟脉冲计数器做递减计数。
- 用循环等待语句查询Stop开关的状态，探测用户何时按下开关做出了反应。
- 当Stop开关按下时，停止计数并计算出经历的时间。
- 将测量出的延时时间转换成BCD数字，并发送到七段显示器上。

微控制器中的各种I/O寄存器的地址如图9-6到图9-9，程序中必须将端口A和端口B配置成图9-19所示的那样。端口A的所有位和端口B的高四位被配置成输出。端口B的低四位中的 $PB_0$ 和

PB<sub>1</sub>用做输入,而PB<sub>2</sub>是一个输出。在两个端口中不需要使用控制信号,因为输入设备是由简单的开关构成的,而输出设备是当驱动器端口引脚上的信号有任何改变时就显示出来的显示器。



536

图9-19 反应计时器电路

首先说明如何用C语言程序实现此任务,然后再说明如何用汇编语言实现。

在两种程序中都有执行完成下列任务的指令,即当Go键被按下时,用计时器实现三秒钟的延时。由于计数器/定时器电路的时钟为100 MHz,计数器要用十六进制数11E1A300做初始化,它相当于十进制数300 000 000。递减计数过程从CTCONT<sub>0</sub>位被置成1时开始。当计数值达到0时,LED打开,同时开始反应时间的测试。接着,计数器设成FFFFFFFF开始反应计时。当测到Stop键被按下时,设置CTCONT<sub>1</sub>=1停止计数过程。总计数值是这样计算的:

$$\text{总计数值} = 0xFFFFFFFF - \text{当前计数值}$$

因为这是时钟周期的总数,按照百分之一秒计算的实际时间是:

$$\text{实际时间} = (\text{总计数值}) / 1000000$$

首先用100去除这个数,将这个二进制整数转换成十进制数,这样生成最高位有效数。余数再被10除,生成下一位有效数,最后的余数是最低位有效数。

### 9.6.1 用于反应计时器的C程序

图9-20给出了一个可能的程序,程序中已将端口A和端口B按要求做了设置,并且已关闭了显示器和LED,该程序轮询在管脚PB<sub>1</sub>上的值。在Go键被按下并且PB<sub>1</sub>变成等于1时,插入一个三秒钟的延时。然后LED被打开并且测试反应计时过程开始。另一个轮询操作用于等待Stop键

537

```

/* 定义寄存器地址 */
#define PAOUT (char *) 0xFFFFFFF1
#define PADIR (char *) 0xFFFFFFF2
#define PBIN (volatile char *) 0xFFFFFFF3
#define PBOUT (char *) 0xFFFFFFF4
#define PBDIR (char *) 0xFFFFFFF5
#define CNTM (int *) 0xFFFFFDD0
#define COUNT (volatile int *) 0xFFFFFDD4
#define CTCON (char *) 0xFFFFFDD8

void main()
{
    unsigned int counter_value, total_count;
    unsigned int actual_time, seconds, tenths, hundredths;

    /* 初始化并行端口 */
    *PADIR = 0xFF; /* 配置端口A */
    *PBDIR = 0xF4; /* 配置端口B */
    *PAOUT = 0x0; /* 关闭显示器 */
    *PBOUT = 0x4; /* 和LED */

    /* 开始测试 */
    while (1) { /* 无限循环 */
        while ((*PBIN & 0x2) == 0); /* 等待Go键被按下 */

        /* 等待三秒钟然后打开LED */
        *CNTM = 0x11E1A300; /* 设置计时器值为300 000 000 */
        *CTCON = 0x1; /* 启动计时器 */
        while ((*CTST AT & 0x1) == 0); /* 等待直列计时器到0 */
        *PBOUT = 0x0; /* 打开LED

        /* 初始化计数器过程 */
        counter_value = 0;
        *CNTM = 0xFFFFFFF; /* 设置起始计数器值 */
        *CTCON = 0x1; /* 开始计数 */

        while ((*PBIN & 0x1) == 0); /* 等待Stop键被按下 */

        /* Stop键被按下，停止计数 */
        *PBOUT = 0x4; /* 关闭LED */
        *CTCON = 0x2; /* 停止计数器 */
        counter_value = *COUNT; /* 读取计数器的内容 */

        /* 计算总的计数值 */
        total_count = (0xFFFFFFF - counter_value);

        /* 将计数转换为时间 */
        actual_time = total_count / 1000000; /* 以1/100秒为单位的时间 */
        seconds = actual_time / 100;
        tenths = (actual_time - seconds * 100) / 10;
        hundredths = actual_time - (seconds * 100 + tenths * 10);

        /* 显示经历的时间 */
        *PAOUT = ((seconds << 4) | tenths);
        *PBOUT = ((hundredths << 4) | 0x4); /* 保持LED关闭 */
    }
}

```

图9-20 用于反应计时器的C程序

被按下。当这个键被按下后, LED关闭, 计数器停止, 并且读出计数器中的内容。就像上面解释的那样完成经历时间的计算, 并将其转换成十进制数。最后的结果是三个BCD数据被整理并写到对应的端口数据寄存器中, 如图9-19中的描述。

### 9.6.2 用于反应计时器的汇编语言程序

图9-21给出了针对反应计时器问题用汇编语言实现的程序。它是基于9.6节开始部分的总体设计原则并仿照图9-20中的C程序编写的。其中对每个指令都给出了注释语句以便于读者可以理解程序的过程。

转换成BCD表示方式需要使用除法指令, 该指令已在2.10.3节中讨论过。这个指令用在寄存器R2中的“被除数”除以寄存器R1中的“除数”。运算结果“商”放在R2中, “余数”放在R3中。注意被100除过以后的实际时间(按1/100秒表示), 可以考虑的只有在R2的商中的最后四位有效数。我们已假定经历时间应该小于10秒, 所以只有三个数字需要显示。商的第一位数字被移到了R4中。其余的数从R3移到了R2中, 并且下次执行除以10的操作。

PAOUT	EQU	\$FFFFFFF1	端口A输出数据
PADIR	EQU	\$FFFFFFF2	端口定向寄存器
PBIN	EQU	\$FFFFFFF3	端口B输入引脚
PBOUT	EQU	\$FFFFFFF4	端口B输出数据
PBDIR	EQU	\$FFFFFFF5	端口B定向寄存器
CNTM	EQU	\$FFFFFFD0	初始计数器值
COUNT	EQU	\$FFFFFFD4	计数器内容
CTCONT	EQU	\$FFFFFFD8	控制寄存器
* 初始化			
	ORIGIN	\$1000	
	MoveByte	#\$FF,PADIR	配置端口A
	MoveByte	#\$F4,PBDIR	配置端口B
START	MoveByte	#0,PAOUT	关闭显示器
	MoveByte	#4,PBOUT	通过使PB <sub>2</sub> = 1关闭LED
* 等待Go键被按下			
GKEY	Testbit	#1,PBIN	Go键与引脚PB <sub>1</sub> 相连
	Branch=0	GKEY	
* 在LED被打开LED之前延迟三秒			
	Move	#11E1A300,CNTM	计时器值为300 000 000
	MoveByte	#1,CTCONT	启动计时器
DELAY	Testbit	#0,CTCONT	等待直至计时器为0
	Branch=0	DELAY	
	MoveByte	#0,PBOUT	打开LED
* 初始化计数过程			
	Move	#\$FFFFFFF,CNTM	设置起始计数器值
	MoveByte	#1,CTCONT	开始计数
* 等待Stop键被按下			
SKEY	Testbit	#0,PB	Stop键连接到PB <sub>0</sub> 引脚
	Branch=0	SKEY	

539

图9-21 使用轮询方式的反应计时器的汇编语言程序

540

* 停止计数并读取最后的计数值		
MoveByte	#4,PBOUT	通过使PB <sub>2</sub> =1, 关闭LED
MoveByte	#2,CTCONT	停止计数器
Move	COUNT,R0	读取计数器
* 计算总的计数值		
Move	#\$FFFFFFF,R2	确定实际的计数值
Subtract	R0,R2	
* 将计数值转换为以1/100秒表示的实际时间, 然后转换成BCD数		
* 将2个最高位BCD数字放入寄存器R4, 最低位BCD数字放入R3		
Move	#1000000,R1	确定1/100秒数
Divide	R1,R2	
Move	#100,R1	除以10得到表示秒数的数字
Divide	R1,R2	
Move	R2,R4	保存这个数字到R4中
Move	R3,R2	将余数作为下一个被除数
Move	#10,R1	除以10得到表示1/10秒数的数字
Divide	R1,R2	
LShiftL	#4,R4	前2个BCD数字放入R4
Or	R2,R4	
* 显示经历的时间		
MoveByte	R4,PAOUT	前2个数字输入到端口A
LShiftL	#4,R3	第3个数字到端口B并保持LED关闭
Or	#4,R3	
MoveByte	R3,PBOUT	
Branch	START	下一次测试准备就绪

图9-21 (续)

表示所用时间的秒数和1/10秒数的BCD数字被存放到了R4中的最低字节中, 并被发送到端口A上去显示。第三个数字表示1/100秒的计数, 它被移到了R3的R3<sub>7-4</sub>位上。由于R3中的此内容要发送到端口B上, 它还需要设置R3<sub>3</sub>=1, 以保证LED是关闭的。

### 9.6.3 最后评价

反应计时器是计算机控制应用的一个非常完整的例子, 与以前各章中的例子相比, 它典型地说明了计算机系统的独立特性。

编写嵌入式系统软件的一个重要特征就是它必须与硬件紧密相关。术语反应系统经常用于描述这样的事实: 在一个时间点上由处理器的外部事件来决定多个程序中哪一个将被执行, 比如关闭一个开关或是在输入端上收到了一个字符这样的事件。就像在第4章中讨论的那样, 有两种机制可以用于协调这些交互过程——轮询和中断。软件设计者必须决定这些交互过程应如何完成。

## 9.7 嵌入式处理器系列

我们已经在9.3节给出了一个嵌入式处理器的例子, 并在9.6节中给出了一个简单应用举例, 现在简要地来讨论一些商用芯片。许多嵌入式应用不需要功能很强的处理器, 显然, 9.1.1节中讨论的微波炉就不需要功能强大的控制器, 因为它需要的计算很简单。针对这样的应用最好是

使用这样一种芯片,它包括一个简单的处理器,但同时还含有足够的内存资源,这样使用这个单独的芯片就可以实现所有的控制功能。在9.1.2节中讨论的数字照相机有很多较高的计算需求,因此它就需要使用一个功能比较强的处理器。

处理器可以用访问内存数据时能够并行处理多少位来标志其性能。目前最强大的微控制器包含有一个32位的微处理器、一个32位宽的数据总线。某些微控制器是基于ARM体系结构建立的。也有可能是某个处理器有一个内部的32位结构,但对于存储器只有16位的数据总线。微控制器Motorola 683xx系列就是这样的例子,它拥有一个基于68000的处理器核。这样的设备被归类为16位的微控制器。最流行的微控制器是8位芯片。它们非常便宜,同时功能足可以满足大量嵌入式应用的需要。还有较小的4位芯片,由于非常简单并且非常廉价,所以也极具吸引力。

[541]

### 9.7.1 基于Intel 8051的微控制器

早在1980年,Intel公司推出了一款称为8051的微控制器芯片。这个芯片具有Intel 8080微处理器系列的基本体系结构,它使用8位芯片,可用于通用计算的应用中。8051芯片得到了快速的普及。它已成为在实际中使用最为广泛的芯片之一。8051有四个8位I/O端口、一个UART和两个16位的计数器/定时器电路。它还包含有4K字节的ROM和128字节的RAM存储单元。同种芯片的EPROM版本中包含有4K字节的EPROM而不是ROM,它被命名为8751。

还有许多基于8051体系结构的芯片;它们进行了不同程度的改进。例如,8052芯片有8K字节的ROM和256字节的RAM,还有一个附加的计数器/定时器电路。它的EPROM版本是8752。

微控制器芯片可以用NMOS或CMOS技术进行构造。CMOS设备有功耗低的优势,因此它们对于使用电池驱动的应用系统非常具有吸引力。以上微控制器的CMOS版本有著名的80C51和80C52。

8051体系结构由Intel公司开发。后来,许多半导体制造商生产的芯片要么是与8051系列完全相同,要么是在性能上做了一些提高但与它完全兼容。从一个嵌入式应用的设计者观点来看,有第二种可用芯片的来源是件好事,因为这样可以保证芯片的高度可用性和价格上的竞争性。

### 9.7.2 Motorola微控制器

在1980年,Intel和Motorola作为半导体芯片制造商,它们占据着主导地位。最流行的8位微处理器成为了微控制器的基础。基于不同的处理器核,有各种各样的Motorola微控制器。

#### 68HC11微控制器

Motorola最流行的8位微处理器是6800和6809。执行的指令集是6800指令集的超集,微控制器芯片称为68HC11。该微控制器有五个可以用于各种目的的I/O端口。I/O结构中包括两个串行接口:一个是异步的,一个是同步的。异步接口使用在10.3.1节中讨论的起止协议。同步接口实现了串行外围设备接口(SPI)。可以将八个模拟输入连接到68HC11上,因为这个芯片具有执行A/D转换的能力。它还有可以完成不同方式的计数操作的计数器/定时器电路。

[542]

在68HC11芯片中包含的存储器总量与具体的型号有关。典型的规格为:在基本型芯片中,有一个8K字节的ROM、一个512字节的EEPROM和一个256字节RAM;在较高级的芯片中,有一个12K字节的ROM、一个512字节的EEPROM和一个512字节RAM。

#### 683xx微控制器

683xx系列的微控制器是基于68000处理器核构成的。这些芯片包含有并行和串行接口、计

数器/定时器以及A/D转换电路。片上存储器总量因芯片不同而不同。比如, 68376芯片有一个8K字节EEPROM和一个4K字节的RAM。

### ColdFire微控制器

68000指令集体系结构为MCF5xxx微控制器——著名的ColdFire嵌入式处理器提供了基础。它们的特殊之处在于使得性能得到大大提高的流水线结构。其实现了完全的32位总线模式。ColdFire处理器核主要是为了用于片上系统(system-on-a-chip)环境而设计的, 这些将在9.9节中讨论。

### PowerPC微控制器

Motorola的高端微处理器是著名的PowerPC, 它是基于RISC体系结构建立的。目前也可以见到以这种处理器体系结构建立的微控制器, 比如包括MPC5xx系列在内的芯片。

## 9.7.3 ARM微控制器

在第3章中给出的ARM体系结构对于嵌入式系统是具有吸引力的, 它具有可靠的计算能力, 并且造价和功耗也相对比较低。它的一个主要目标是使得ARM处理器的设计适合于片上系统(system-on-a-chip)的环境。ARM微控制器也可以作为一个单独的芯片使用。

ARM处理器核已经有了一系列用于嵌入式应用的产品, 包括ARM6、ARM7、ARM9和ARM10。基本的ARM结构使用32位的体系结构, 并且其中的所有指令是32位长的指令集。还存在另一种版本, 称为Thumb(拇指)型, 它使用16位的指令和16位的数据传递。Thumb版本使用的是ARM指令的子集, 该子集被编码并嵌入到16位的格式中。它包含有比ARM体系结构少一些的寄存器。Thumb的优势是在大多数情况下只需要使用相当小的存储器来存储程序, 因为这些程序是使用高度编码的16位指令构成的。在执行时, 每个Thumb指令被扩展成正常的32位ARM指令。这样, 一个Thumb型的ARM核中除了正常电路外, 还包含有一个Thumb解压缩装置。

ARM体系结构和处理器核已经由ARM公司开发出来, 许多其他的公司也被授权提供这种核。一些公司像Atmel公司、Sharp Electronics公司及Samsung半导体研究所也生产基于ARM核的微控制器芯片。例如, Atmel的AT91F40416微控制器使用ARM7-TDMI Thumb-aware核, 它还包含有4K字节的RAM、526K字节的闪存(Flash ROM)、32位可编程I/O线、2个串行口和1个计数器/定时器电路。

## 9.8 设计问题

一个嵌入式系统的设计者必须做出许多重要的决定。对实际应用或是将要设计的产品的特性提出了特定的要求和限制, 在本节中我们将考虑设计者要面对的一些最重要的问题。

### 成本

在许多嵌入式应用中电路的成本必须是低廉的, 廉价的解决方案体现在是否能够用单个微控制器芯片实现所有必须提供的功能。当有足够的I/O能力和有充足的片上存储器来存放需要的软件时, 做到这一点是可能的。

### I/O能力

微控制器芯片提供多种I/O资源。这个范围包括从简单的并行和串行端口到计数器、定时器、A/D及D/A转换电路等扩展性支持设备。

可用的I/O线数是重要的, 没有足够的I/O线时需要使用外部电路去补充它。这些已经在图

9-19的反应计时器中作了说明,其中扩展译码器电路用于驱动由微控制器提供的4位BCD信号的七段显示器。如果微控制器有四个而不是两个并行端口,那么就可以将每个七段显示器与一个端口相连。控制程序可以更直接地生成显示过程中驱动各个段所需要的输出信号。

### 规格

微控制器芯片有各种不同的规格,如果一个应用中可以用8位微控制器做处理,就没有必要使用16位芯片,否则它就会变得价格比较昂贵、实际尺寸增大并且功耗加大。大部分的实际应用可以使用相对较小的芯片进行处理。最近这些年芯片销售量最大的是8位类型,紧随其后的是4位和16位类型。

就芯片占用印刷电路板面积而言,芯片的实际大小很重要。这个面积是所含费用中的主要部分。

### 功耗

功耗在所有的计算机应用中是一个需要重点考虑的内容。高性能的系统其功耗也会变得很高,需要增加一些机制来驱散产生的热量。在许多嵌入式应用中消耗的能量很低,所以不用担心散热问题。但是,这些应用通常是电池电源产品,所以电池的寿命(根据功耗决定)是主要的因素。

544

如果使用CMOS技术构造微控制器芯片,其功耗就会降低。在CMOS技术中,功耗与时钟频率成正比。如果低性能可以满足指定的应用,就可以降低时钟频率来降低功耗。另一种可能采用的折中方法是与微控制器芯片的功能有关的,降低功能意味着减少芯片上的电路,这样也就降低了功耗。

### 片上存储器

微控制器芯片中包含的内存使得简单嵌入式应用可以使用单个芯片来实现。存储器的大小和类型有很大的差异。相对少量的RAM可能对于计算中的数据存储是足够的。大量的只读存储器可用于存储程序,这种存储器可以是ROM、PROM、EPROM、EEPROM或闪存,它们的价格依次递增。对于大容量的产品,最为经济的选择是采用带ROM的微控制器。但是,这也是一种最不灵活的选择,因为ROM中的内容是在芯片制造时被永久实现了的。PROM和EPROM型存储器可以在嵌入式产品制作时进行编程。最灵活的应用是由EEPROM和闪存提供的存储方式,它们可以被多次编程。

对于计算要求比较高的应用,需要使用外部存储器。有些微控制器不包含任何的片上存储器,它们通常用于存储总量非常大,不能在微控制器芯片内部实现的复杂应用中。

### 性能

当微控制器在家用设备和玩具中使用,除了像索尼PlayStation中的电视游戏以外,性能通常不是个大问题。在这种情况下可以选择小型并且便宜的芯片。但是,在移动电话和掌上游戏机的应用中就需要有非常高的性能。高性能就需要比较强大的芯片,这也必将带来高费用和较大的功耗。由于这些应用通常是用电池做电源的,降低功耗是最为重要的。在第3章中我们讨论了ARM体系结构。实现这种结构的一种范例是StrongARM芯片,它专门被设计成低功耗且有良好性能的产品。9.7.3节中讨论的ARM体系结构的Thumb版本,其目的是为了用于嵌入式应用,在这些应用中成本和性能问题是关键。

### 软件

使用高级语言编写计算机应用程序有许多优势,它们使程序开发过程变得简单并且使将来



的软件维护和修改工作更加容易。但是,在有些情况下求助于汇编语言可能会更加理想。使用汇编语言编写的程序可能生成的目标码会比由编译程序生成的代码压缩10%~20% (就存储需要的总量来讲)。如果一个嵌入式应用是基于具有片上存储器的微控制器而建立的,那么可以将所需的代码放进内部存储器中,而避免使用外部存储器,这就成了使用汇编语言的主要优势。

[545]

设计者应该仔细而恰当地估计所得到的片上RAM的容量。这种存储器通常用于存储动态数据,就像一个临时缓冲区,或是用于实现堆栈。编写看似紧凑的代码是容易做到的,例如使用C语言,但是它需要的RAM比可用的要多一些。

#### 指令集

另一个重要的问题是处理器所使用指令集的特性。CISC类指令比RISC类指令产生更多紧凑型的代码。因此,处理器的选择对代码的大小也有影响。由ARM体系结构的Thumb版本提供了一个有趣的例子,为32位处理器设计的RISC类指令集已经修改成了用于16位指令的高度编码集。为Thumb版本编写的程序与用完全ARM体系结构编写的程序相比,其达到了30%的紧缩效果。指令在执行时被恢复,Thumb指令被扩展成正常的ARM指令,就像在9.7.3节中解释的那样。

#### 开发工具

数字系统的设计者对开发工具的依赖性很强。这些开发工具包括计算机辅助设计(CAD)软件包、操作系统软件、编译程序、汇编程序及处理器的模拟器。开发工具的范围和可用性通常依赖于所选择的嵌入式处理器。有第三方支持也是非常具有吸引力的,因为在第三方那里有工具和文档资料。有效的文档和来自于制造商的建议(如果需要的话)是非常宝贵的。

#### 可测试性和可靠性

印刷电路板通常是很难测试的,尤其在高密度地使用芯片时更是如此。如果将整个系统设计成容易测试的方式,那么测试过程就会大大简化。一个微控制器芯片可以包含一些电路,这些电路可以使包含该芯片的印刷电路板比较容易测试。例如,在有些微控制器中包含有一个测试端口,它与用于可测试结构的IEEE 1149.1标准兼容,该标准是以测试访问端口及边界扫描结构<sup>[1]</sup>而闻名的。

嵌入式应用要求具有稳健性和可靠性。一个典型产品的生命周期希望最少可达到5年以上。这一点与个人计算机不同,个人计算机容易在短期内被淘汰。

## 9.9 片上系统

在一个嵌入式应用中,尽量使用少量的芯片是可取的。理想的情况是单个芯片可以实现整个系统。在比较简单的应用中,用那些可获得的商用微控制器实现所有需要的功能是不可能做到的。但是在比较复杂的应用中就不行了。有些微控制器芯片以特殊应用为目标,这些特殊应用很难使用通用微控制器去实现。例如,一个用于视频游戏的微控制器应该包含视频和声音处理电路。这些需求与用于激光打印机或是移动电话中的微控制器是完全不同的。

[546]

开发一个复杂的微控制器是一件需要花费时间且具有挑战性的任务。但是,大部分消费品的开发时间又必须要尽量短。如果设计者可以利用一些已有的电路模块,而这些模块可以得到并且易于使用,那么一个为了特殊应用而实现的完整系统芯片就可以在相对短的时间里设计出来。微处理器核是所需模块之一。这些核可以通过一个使用权限许可来获得。其他的用于实现存储、A/D和D/A转换的电路或DSP(数字信号处理)电路的模块也可以得到。设计者可以通过使用这些模块设计出其余所需的电路并完成设计。

在9.7.3节中,我们曾提到ARM的核已经设计成大型系统的一个模块。另一个有趣的例子是National Semiconductor公司的CompactRISC核,它的一个特性是具有从8位到64位的可伸缩性。它有一个简单的3段流水线、一个40K字节ROM和1.4K字节RAM的片上存储器。只有当还需要外部存储器时才添加一个总线接口单元。这样,核的复杂性可以调整,并与应用要求的功能相适应。

系统核和其他模块的提供者是在销售他们的设计而不是芯片,实际上,他们是在销售一种思想而不是实际的部件。其产品是典型的知识产权(IP)的例子,这些产品可以被其他人用来设计自己的芯片。

## FPGA的实现

现场可编程门阵列(FPGA)提供了一种在单个芯片上实现系统的有效手段。它不是像微处理器芯片那样为设计者提供一个预定好的功能单元,FPGA设备在设计阶段可以完全自由地进行。它们可以很容易做到包含某个标准单元,然后按照需求构造系统的其余部分。为了说明这种方法的典型特点,我们将对Altera Corporation的Excalibur系统进行分析。

FPGA功能性能能力已经有了突飞猛进的增强。单个大型FPGA芯片可以实现一个需要数十万个逻辑门来完成的系统。这样的芯片已经足可以满足微控制器和在系统所需要的其他电路实现的典型功能。

任何一个片上系统的核心构件是处理器核。Excalibur系统给出两种不同的选择:一个是相关的处理器,它是用软件定义的;另一个所涉及的处理器为FPGA芯片,它是制造过程中在硅片上实现的处理器核。

### 软处理器核

Excalibur系统提供一个软件模块,它用Verilog硬件描述语言写成,所实现的处理器体系结构称为Nios。这种体系结构允许设计者将处理器描述成库中的任意一个模块,这些库中的模块可以用两种方式完成,一种是硬件描述语言比如Verilog或VHDL,或是在原理图输入时用功能块完成。设计者可以根据系统的性能需要来选择32位或16位的处理器。

547

9.3.1节中描述的并行接口模块是一个有效的参数化库模块。用户可以具体指明参数,使其适合于设计的要求。寄存器的长度可以在1到32位范围内选择。用户可以选择完全双向的接口或是有一定限制的接口方式。例如,可以指定仅作为输出的端口,在这种情况下只指明了输出数据寄存器,而输入通道和数据定向寄存器是不生效的。这样使得FPGA的资源不会浪费在不需要的部件中。

串行接口可以用UART电路的形式获得。设计者指明所需的参数,例如数据位的位数、停止位位数以及使用哪种奇偶校验位。从预定义的标准范围内选择传送/接收时钟频率。这一频率可以在以后根据应用软件需要用一个内部的除数寄存器来改变,该寄存器可以装入用来划分时钟频率的值。用户仍然可以只选择需要的功能并只实现相应的电路。

定时器组件提供的计数和定时能力在9.3.3节中作了描述。它的操作完全由应用软件进行控制。

大型FPGA芯片包含相当大量的存储器,这些存储块可以用于实现当存储器容量需求不太大时嵌入式系统中的RAM和ROM部件。设计者可以按照字数和每个字的位数说明需要内存的容量。如果片上存储资源不够,可以说明一个外部存储接口,这将导致在FPGA芯片的引脚上实现相应的存储总线信号。

ExcaliburCAD工具使得在FPGA上设计系统变得更加容易。该工具中包含一个向导程序,它

提示设计者输入需要的参数,然后为设计者生成指定电路。因此,处理器和I/O模块可自动实现。它们被一个类总线结构(用来实现Nios总线协议)相互连接着。我们将注意到FPGA芯片上的总线结构不是使用三态驱动器实现的,如同第7章中描述的那样。FPGA是一个通用装置,它包含有大量的逻辑单元、连接线路和开关。三态驱动器仅适合于用作特定的用途,因此在一般的FPGA中不提供。使用多路复用电路可以实现三态总线的功能。取代单个双向路径,在每个方向上都使用了独立的多路复用器。尽管这种方法需要用许多门,但是这是一种灵活的方案,因为

548 所需的逻辑单元和互连的资源仅仅是FPGA总资源中的一小部分。

处理器和接口子系统占用了FPGA芯片相对小的一部分,芯片的其余部分可以从用于专门应用电路的实现中获得。这些电路可以直接连到处理器的总线上或是使用更加灵活的专用I/O端口与其连接。在一个片上系统环境中,处理器子系统的I/O端口不需要连接到FPGA设备的I/O引脚上。取而代之的是,设计者可以用它去连接一个专用的应用电路,该电路是在处理器系统中用FPGA实现的。

Nios处理器有一个类RISC的指令集。它的运行能力可以达到50MIPS(每秒百万指令数)。设计者可以选择在同一个FPGA上实现多个Nios处理器,这样可实现一个多处理器系统。

#### 硬处理器核

软处理器核的一种替代方法是在硅片上实现处理器,这样可构造一个专用的FPGA。Excalibur系统提供基于不同处理器的此用途的FPGA。其中一个例子是在其中一部分实现了ARM处理器核的FPGA。除了处理器电路外,ARM处理器总线、RAM模块及UART系列模块也在硅片上得以实现。这样可以实现相对高性能的系统。芯片的其余部分由一般的FPGA资源构成。使用硬处理器核,系统的运行性能可达数百MIPS。

#### 设计者的观点

嵌入式系统的设计者肯定要寻找最简单和最有成本效益的技术途径。微控制器芯片对于那些必须要实现一个完整系统的需求来讲可能是一种最好的选择。如果需要用附加芯片的方式实现系统,情况就不同了。这时,FPGA解决方式就变得更具吸引力,因为它们可能只需要较少的芯片就可以实现这个系统。

另一个要考虑的内容是预设计模块的可用性。一个微控制器芯片包含了许多不同的模块,任何使用这些模块不能实现的特性,必须使用附加芯片来实现。一个FPGA设备允许设计者设计任何类型的数字电路。许多实际的设计中包含许多执行通用任务的电路,这样的电路应该能从模块库中获得。使用I/O接口和定时器电路的情况就是一个典型的例子。如果其他的有用模块可用,那就非常方便了。对于信号处理的应用,库中应包含有专用的过滤器电路和快速乘法器。如果设计的系统需要通过一个标准总线(如PCI)连接到其他的计算机上,那么如果PCI接口可以作为一个预设计模块得到,设计者的任务是非常简单的。

## 9.10 结束语

这一章以一个简单的应用介绍了一个完整的嵌入式计算机系统设计过程。我们没有使用特殊的微控制器,因为其中包含的原则是一致的。它们是嵌入式系统设计者要面对的核心问题。

549

理解硬件和软件间密切的相互作用是非常重要的。设计可能包括对轮询I/O和中断的权衡、不同的指令集在功能和代码压缩间的权衡、功耗和性能间的权衡,等等。

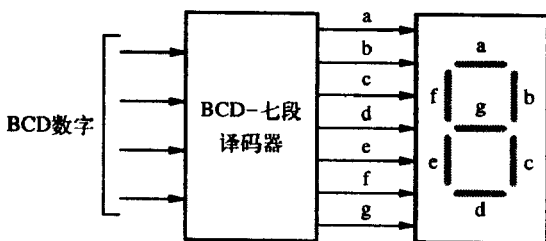
已经非常庞大并仍在迅速扩展的嵌入式应用领域为计算机技术的创造性应用提供了巨大的

机会。许多书中都介绍了嵌入式系统<sup>[2-4]</sup>这一焦点应用。

## 习题

- 9.1 在9.3节中给出的微控制器接收十进制数(0到9),将它们换成ASCII编码字符送到串行口上。每当一个数字到达时,它必须被显示在连接到并行端口A上的七段显示器上。给出连接中所需要完成的功能。显示器的段标志如图A-33所示。写一段C语言程序执行要求的任务。用轮询法检测每个ASCII码字符是否到达。
- 9.2 写一段汇编语言程序实现9.1中的任务。
- 9.3 使用中断方式检测每个ASCII码字符是否到达来解决9.1中的问题。
- 9.4 针对9.3中的问题写一段汇编语言程序。
- 9.5 9.3节中的微控制器在它的串行口上接收十进制数。每个数字由两个ASCII码字符的数字编码组成,为了区分连续的两位数字,使用H作为定界符。这样如果两个连续的数字是43和28,接收的序列将是H43H28。每个数字将被显示在连接到端口A和B上的两个七段显示器上。定界符不应该被显示出来。只有当下一轮接收到两个数字时显示的内容才会被改变。给出这种连接所要完成的功能。显示器的段标志如图A-33所示。写一段C语言程序执行所需完成的任务。使用轮询方式检测每个ASCII码字符是否到达。
- 9.6 写一段汇编语言程序实现9.6中的任务。
- 9.7 使用中断方式检测每个ASCII码字符是否到达来解决9.5中的问题。
- 9.8 针对9.7中的问题写一段汇编语言程序。
- 9.9 在9.3节中微控制器使用它的串行口接收十进制数,每个数字由四个ASCII码字符的数字编码组成,为了区分连续的四位数字,使用H作为定界符。这样如果两个连续的数字是2143和6292,接收的序列将是H2143H6292。每个数字将被显示到四个七段显示器上。假设每个显示器上有一个BCD-七段译码器电路与其相连,如图P9-1所示。给出到微控制器的必要连接,写一段C语言程序执行需要完成的任务,并使用轮询方式检测每个ASCII码字符是否到达。

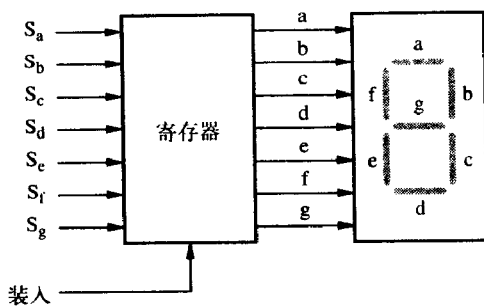
550



图P9-1 使用BCD译码器的七段显示器

- 9.10 写一段汇编语言程序实现9.9中的任务。
- 9.11 在解决9.9中的问题时,使用中断方式检测每个ASCII字符是否到达。
- 9.12 写一段汇编语言程序实现9.11中的任务。
- 9.13 重复9.9题,但假定每个七段显示器有一个7位寄存器与其相连,而不是BCD-七段译码器。该寄存器有一个控制输入端Load,当Load=1时,7个数据位被装入到寄存器中。寄存器中的每一位可以驱动相连显示器的一个段。图P9-2给出了寄存器显示器方案。微控制器的输

出连接在并行端口A上, 该端口为所有的四位显示器提供数据。



图P9-2 使用一个寄存器的七段显示器

- 9.14 重复9.13中的问题, 使用汇编语言编写程序。
- 9.15 解决9.13中的问题, 使用中断方式检测每个ASCII码字符是否到达。
- 9.16 写一段汇编语言程序实现9.15中的任务。
- 9.17 在9.5节中我们假定源设备在脉冲串中最多产生80个字符。当允许产生字符多于80个字符时, 图9-17和图9-18中的程序是否还合适? 如果不行, 给出对该程序的修改程序。
- 9.18 在图9-17的程序中, 确定哪个环形缓冲区为空的测试方法是检查fin和fout的变址值是否相等。可以引入一个计数器变量M来替代这种检测方式, M表示缓冲区中当前的字符数。使用这种方式修改该程序。
- 551** 9.19 对于图9-18中的程序重复9.18中的问题。
- 9.20 修改9.6节中的反应计时器, 假定被测试人总是在一秒之内做出了反应。这样经历的时间应该用两位数字显示, 其表示一秒的百分之一。将两个七段显示器连接到端口A上并且修改图9-20和图9-21中的程序, 实现所需要的操作。
- 9.21 在图9-19中, 每个数据的七段显示器都包含着一个BCD-七段译码器; 因此微控制器对每个将要被显示的数字同时要提供一个4位BCD码。假设将它用译码器代替, 每个七段单元有一个含有控制输入Load的7位寄存器, 当Load = 1时, 这7个数据位被装入到寄存器中。寄存器中的每一位驱动连接到显示器上的一个段上。图P9-2给出了寄存器显示器方案。扩展图9-20和图9-21中的程序使其能够适用于这个寄存器显示器电路。
- 9.22 在图9-21中, 一个按照百分之一秒表示反应时间的二进制数, 使用被100和10连除的方法转换成等量的BCD数字。实现这种转换的另一种方法是被10连除, 在这种情况下, 每次除的余数是一个所需的BCD数。用这种方法产生的数字顺序是怎样的? 修改图9-21中的程序, 执行这种转换。
- 9.23 在9.3节中用微控制器生成了一个“日计时”时钟。这个时间(小时和分钟)被显示到四个七段显示器上。假定每个显示器有一个BCD-七段译码器与其相连, 如图P9-1所示。同时假设使用一个100MHz的时钟。给出需要的硬件连接并写出相应的程序。
- 9.24 重复9.20中的问题, 但假定每个七段显示器有一个寄存器与其相连, 如同图P9-2中所示的那样。
- 9.25 在一个用单芯片实现的系统中, 处理器和主存储器驻留在同一个芯片。在这个系统中是否需要高速缓存? 请解释。

## 参考文献

1. *Test Access Port and Boundary-Scan Architecture*, IEEE Standard 1149.1, May 1990.
2. W. Wolf, *Computers as Components — Principles of Embedded Computing System Design*, Morgan Kaufmann Publishers, San Francisco, CA., 2001.
3. K. Hintz and D. Tabak, *Microcontrollers*, McGraw-Hill, New York, 1992.
4. J. B. Peatman, *Design with Microcontrollers*, McGraw-Hill, New York, 1988.



# 计算机外围设备

### 本章目标

在本章中你将学习以下内容:

- 计算机输入和输出设备是如何工作的
- 扫描仪和打印机的操作
- 图形卡和图形图像处理
- 同步和异步串行数据连接
- 使用ADSL和电缆调制解调器的高速Internet连接

553

在前面的章节中我们讲述了处理器、存储器、磁盘和只读光盘存储器（CD ROM）的硬件和软件特性，还讲述了计算机与外部设备进行通信的方法，包括支持程序控制输入输出、中断和直接存储器访问的硬件和软件。本章主要介绍一些常用的计算机外围设备以及它们在计算机系统上的连接。

外围设备是指与计算机相连的任何外部设备。这里所说的计算机只包括处理器和它的存储器。按功能划分，计算机外围设备可分为两大类。第一大类包括执行输入输出操作的设备，如键盘、鼠标、跟踪球、打印机和视频显示器等。第二大类包括主要用于数据辅助存储的设备，主存储由计算机主存储器提供。一些大容量的存储设备，尤其是磁盘，被用作数据的在线存储。其他的，如光盘、软盘和磁带，这些存储介质可以与驱动部件分开，从而可以将数据从一个计算机系统传输到另一个计算机系统。例如光盘是发布软件最常用的存储介质，也被称为只读光盘存储器（CD ROM）。辅助存储设备在第5章已经讨论过。

当今，最重要的计算机外围设备是那些提供与Internet连接的设备。近些年，计算机领域飞速增长都是计算机与通信相结合以及大量新技术应用于网络的结果。这些发展涉及到我们生活的每一方面，从商业到娱乐、教育。

在这一章，我们将概括介绍现代计算机系统输入输出设备的种类，并简要描述其中包含的技术。机箱以外的设备常常是使用串行方式与处理器相连，可以是有线的或无线的。本章会介绍一些串行通信的基本概念。

### 10.1 输入设备

输入设备包括键盘和用来移动屏幕光标的设备，如鼠标、跟踪球以及操纵杆等。扫描仪和数字照相机也广泛用来获取图像，然后以数字化形式输入计算机。



### 10.1.1 键盘

最常遇到的输入设备是键盘，通常附带有鼠标和跟踪球，与输出设备视频显示器一起用来进行直接的人机对话。

键盘有两种类型。一种是由安装在印刷电路板上的机械开关阵列组成的。这些开关按行列组织并与电路板上的微控制器连接。当一个开关被按下时，控制器识别出它的行列位置确定是哪个键被按下了。确认反弹后（参见第4章），控制器就产生该键的代码并通过串行连接发送给计算机。

554

另一种键盘使用三层平板结构。顶层是塑料材质，表面上标记着键的位置，下侧安装有导线。中间层是用橡胶制成的，对应键的位置有个小洞。底层是金属材料，对应键的位置处有凸块。当在顶层的键位置上施加压力时，顶层下侧的导线与底层相应的凸块相接触，这样就与机械开关一样形成一个电流回路，回路中的电流将被微控制器检测到。采用这种结构，我们可以得到一个低价、同样结实耐用并可防止食物与饮料污染的键盘。在销售终端的应用中常常可以见到这种类型的键盘。

### 10.1.2 鼠标

1968年鼠标的发明是人机对话发展的一个重要阶段。在那以前，文本是主要的数据输入形式。鼠标使得通过绘制想要的对象来直接输入图形信息，并引入了许多新的、有效的概念，包括窗口和下拉菜单。

鼠标的形状非常符合操作者的手形，这样在一个平整的表面上可以随便地移动它。一个电路将检测到该移动信息，并将X和Y方向移动距离的测量值发送给计算机。移动检测既可以是机械的也可以是光学的。机械式鼠标安装有一个小球，小球随鼠标自由滚动。通过检测小球的滚动来驱动两个计数器计数，每个计数器对应一个方向的移动。鼠标还安装有2到3个按钮。微控制器收集计数器和按钮信息，然后编码成一个3个字节的数据包通过串行连接发送给计算机。

光学鼠标使用一个发光二极管来照射放置鼠标的表面，用一个光敏元件检测从表面反射回来的光。有一些类型的光学鼠标必须在专用的垫子上滑动，这种垫子上有许多水平和垂直线条组成的图案。反射光随着鼠标在平面上明暗区域上的移动而变化，通过计算这些变化测定鼠标移动的距离。

555

智能鼠标是一种很复杂的鼠标，于1999年由Microsoft公司推出。它几乎可以在任何表面上使用。取代简单的光传感器，鼠标下的平面的一个小区域图像被采集到一个微小的数字照相机中，将图像信息数字化。该数字照相机每秒钟可以采集1500张图片。除非表面是完全一致和光滑的，比如镜子，否则它的图像就会有线条、明暗变化等特征。通过比较这些连续的图像，鼠标内置处理器就可以相当精确地测量出移动的距离。该处理器采用关联信号处理技术来确定从一个图片到另一个图片的位移。这是一项计算密集型任务，每秒钟必须要重复计算1500次，所以必须有功能强大并且廉价的嵌入式处理器才可行。这种处理器每秒钟能执行1800万条指令。

自从鼠标被发明后，又开发出了许多具有相同功能的设备，如跟踪球、操作杆和触摸垫。

### 10.1.3 跟踪球、操作杆和触摸垫

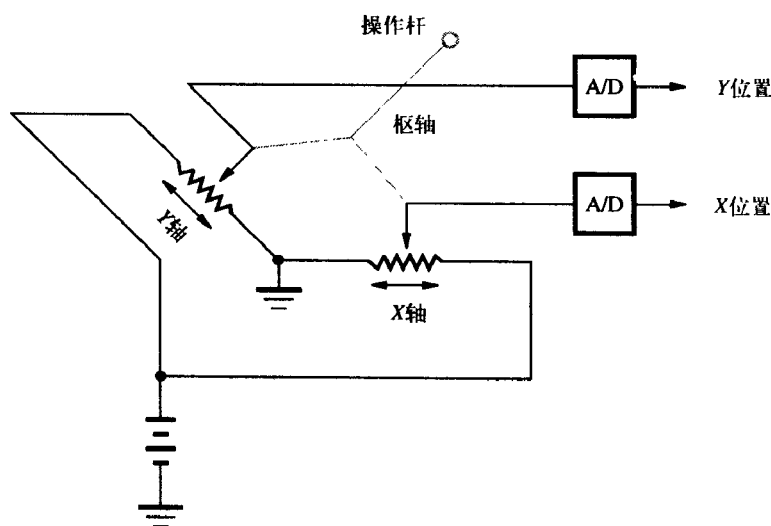
鼠标可以使操作者能够移动屏幕上的光标。为了适应各种各样的应用环境和使用者的爱好，

又开发出了大量具有相似功能的新型输入设备。

跟踪球的工作原理与机械式鼠标十分相似。将一个小球安装在键盘上，使用者通过滚动小球来控制屏幕上光标移动。

操作杆是一根装有枢轴的短杆，用手可以将它在X-Y平面上进行任何移动。当这一移动信息发送到计算机后，相应的软件就将屏幕上的光标沿相同方向移动。

操作杆的位置可以被合适的线形或角形位置传感器检测到，如图10-1所示的电压计装置。电压计X和Y的电压输出分别输入到两个模/数（A/D）转换器上，转换器的输出决定操作杆的位置，从而决定移动的预期方向。



556

图10-1 操作杆，采用电压计作为位置传感器

操作杆常应用于笔记本电脑和视频游戏上。应用于笔记本电脑时，操作杆安装在键盘上的按键之间并稍高于按键。由于它的安装位置，操作杆具有不需要操作者将手移开键盘就可以进行操作的优点，用一个手指轻推它就可以定位屏幕上的鼠标。操作杆非常结实耐用且占用空间小。为了用在视频游戏中，操作杆被设计成一种符合游戏本身特性的手柄，并常常装配有按钮控制用于发射一个球或开枪。

另外一种非常有用的输入设备是触摸垫，以及和它类似的设备——触摸屏。触摸垫是用压力敏感材料制成的小垫。当使用者的手指或笔尖触及到垫子上的某一点时，压力使该点的电性质发生变化，这样该点的位置就被检测出并传送给计算机。只需在垫子上移动手指，使用者就可以指示软件在相同方向上移动屏幕上的光标。这就使得触摸垫可以成为鼠标或操作杆的低价取代设备，并且由于它没有任何移动部件，所以非常结实耐用，并具有高可靠性。触摸垫非常适用于便携式计算机。

许多新型材料已经被开发出来制作触摸垫。其中最具有创新意义的可能是嵌入了大量微小光纤的材料。这种材料不仅能够识别与它相接触的物体的位置，而且也能识别施加压力的大小。这种材料主要是为太空机器人应用而开发的，但很快就发现了其他应用，如作为一种输入设备取代并扩充钢琴键。

触摸垫与液晶显示器结合起来生产出一种触摸屏，既可以用作输入又可以用作输出。这种

屏幕普遍用于个人数字助理(PDA)中,比如Palm Pilot。另外,还有一种使用阴极射线管的触摸屏,手指触摸这种屏幕引起的电容变化在电子束扫描屏幕显示图像时被检测到。触摸屏常用于收银机和销售终端点上。

#### 10.1.4 扫描仪

扫描仪是将印刷材料和照片转换成数字信息的设备。早期的扫描仪其扫描页被安装在绕着传感器转动的滚筒上。今天的扫描仪大都是平面结构,扫描页铺放在平坦的玻璃表面上。光源扫过页面,反射光被聚集到电荷耦合器(CCD)的线性阵列上。当电荷耦合器件感光时,一个电荷就被储存在与之相连的微小电容器中,这样电荷量就和光强成比例。这些电荷将被适当的电路收集起来,并通过一个模/数转换器转换成数字形式。对于彩色扫描仪,红色、绿色和蓝色滤光器用来分离三种基本色,然后分别进行处理。当光源扫过页面时,传感器阵列重复读取,从而取样出图像的连续像素线。需要指出的是这种技术也用于数字复印机,数字复印机实际上就是扫描仪和激光打印机的结合产物。

[557]

将一张印刷页扫描到计算机后,图像在存储器中以像素阵列的形式表示出来。最简单的情况下,一个像素用一位表示,来显示图像中相应点是明还是暗。对于更高品质的图像,每个像素需要存储更多的信息来表示相应点的颜色和光强。每个像素可以使用长达3个字节的信息,分别对应三种原色中的一种。

考虑一页文本的例子,图像中的黑暗区域对应于印刷的字符。许多字符识别技术已经被开发出来,可以通过分析存储在存储器中的像素图来识别图像中的字符。因此,就可以创建一个描述印刷页内容的文本文件,在这个文件中每个字符被转换成适当的二进制代码,如ASCII码。这样结果文件就可以被文本处理程序如Word处理了。

### 10.2 输出设备

计算机输出可以采取各种各样的形式,包括字符文本、图形图像和声音等。下面我们讲述一些常用的输出设备。

#### 10.2.1 视频显示器

在任何需要可视化显示计算机输出的时候,都要用到视频显示器。其中最常用的是使用阴极射线管(CRT)显示器。

首先讲述一下如何在阴极射线管显示器上形成图像。聚焦后的电子束撞击到荧光屏上后引起发光,在黑暗背景衬托下看起来就像一个亮点。当电子束熄灭或转移到其他点时,这个亮点就消失了。因此,一般来说,有三个独立的变量随时都需要指定值,来表示电子束的位置和强度。电子束的位置指的是屏幕上点的X和Y坐标。电子束的强度,通常称为Z轴控制,提供该点的灰度和亮度信息。屏幕上可寻址的最小区域称为像素,它由许多大小不同的小点按照某些几何图形排列而成。按不同的组合方式照亮这些点可以得到不同级别的亮度。这种技术称为半色调。在彩色显示器中,每个像素有三种不同颜色的荧光点:红色、绿色和蓝色。按不同的组合方式照亮这些点可以得到不同的颜色。

[558]

电子束在屏幕上形成的点的大小决定图像中总像素的多少。沿X或Y轴通常都有700~2500个像素。Z轴信息可以在24位内进行描述,每个字节对应于一种颜色。它能够产生用人眼所能观察

到的最高色彩分辨率。计算机视频显示器的最通用标准是VGA（视频图形阵列）以及它的高品质升级版本。基本的VGA显示器有 $640 \times 480$ 个像素。升级标准规定显示器有更高的分辨率，如 $1024 \times 768$ （XVGA）和 $1600 \times 1200$ （UXGA）。

采用光栅扫描技术，字符文本和图片都可以被显示出来。电子束连续地从左到右扫描每一行像素，从上到下，直到扫描完屏幕上所有行。许多显示器都采用隔行扫描以提高观察到的屏幕刷新速率。在这种扫描方式中，电子束扫描屏幕两遍，一遍扫描奇数行，另一遍扫描偶数行。将要显示的图像被存储在一个显示缓冲区中，这个缓冲区在扫描期间提供Z轴信息。在最简单的表示方式中，用一个位图来描述屏幕上将要显示的图像，位图中每一位对应一个像素。因此，一个有 $1024 \times 1024$ 个像素的屏幕就需要1Mb的显示缓冲存储空间。为了能以60次/秒的速率刷新显示器，数据传输速率要达到60Mb/s。今天的高品质显示器每个像素要使用32位，因此需要的显示缓冲空间和通信带宽就更大。通常，在32位中只有24位用来存储色彩信息，第4个字节主要用来与主处理器的字长相兼容，同时也为日后的升级提供空间。现代的计算机系统能够在屏幕上重叠多幅不同的图像（如在基于窗口的操作系统中），因此就需要几个独立的显示缓冲区。

### 10.2.2 平面显示器

在计算机显示应用中，虽然阴极射线管技术占主导地位，但平面显示器却越来越来流行。平面显示器更薄更轻，能提供更好的线性，在某些情况下甚至有更高的分辨率。多种类型的平面显示器已经被开发出来了，包括液晶显示器、等离子显示器和场致发光板显示器。低价平面显示器的出现十分有助于笔记本电脑的发展。

液晶显示器是由一薄层液晶——一种具有水晶性质的液体——夹在两块透明板之间构成的。顶板上安装有透明的电极，底板是一面镜子。通过施加适当的电信号来穿过两块板，将液晶的不同部位激活，从而改变这些部位的光散射性和极性发生变化。因此这些部分既可能传导光线也可能阻断光线。通过液晶中选定部分的光线形成一幅图像，然后镜子将其反射回来给观察者。液晶显示器常应用于手表、计算器、笔记本电脑以及其他设备中。

559

液晶显示器有两种类型。静态显示器的结构比较简单，它里面的电极分别沿着顶板中的一个轴和底板中与该轴垂直的轴安装，这样就定义了列和行。要照亮某一特定部分，需要在一个行电极和一个列电极之间施加一个电压。这个电压产生一个带电区域使两个电极交点处的液晶区域被点亮，显示出一个亮点。这种结构的显示器制造容易且价格便宜，但图像质量较低。由于照亮的区域没有很好地被界定，因此图形的边界也不是很清晰。同时，由于长电极的电容很大，点亮和熄灭的速度就比较慢。例如，当鼠标在屏幕上快速移动时，迟缓的反应就会使鼠标后面拖有一个尾巴。

通过在每一交点处引入一个晶体管可以得到更高品质的显示器，这种显示器反应速度快，并且能够更好地控制需要被照亮的区域。晶体管安装在一块薄膜中，这块薄膜安装在其中的一个板中，因此这种类型的显示器称为薄膜晶体管显示器（TFT），也称为有源矩阵显示器，常用于高档笔记本电脑中。

等离子显示器是由两块玻璃板组成，玻璃板之间被一薄层充满氖气之类的气体隔开。每块板都有几个横穿它们的平行电极。两块板上的平行电极都是在右角到达另一块板上。将一个脉冲电压施加到属于不同板的两个电极之间，就会使两个电极相交处的一小块区域的气体发光。

并且由于有持续施加在所有电极上的低电压,那部分气体将持续发光。同时还有一个相似的脉冲装置用来选择性地熄灭亮点。等离子显示器能够提供很高的分辨率,但价格昂贵,通常用于那些对显示质量要求非常高,且不宜使用大体积的阴极射线管的场合。

场致发光板显示器在两块电导板之间加入一薄层磷,当电信号到电导板上时就会导致磷发光而形成图像。

平面显示器在各种应用中的生存能力与相竞争的阴极射线管显示器技术的发展有着密切的联系,至今阴极射线管技术仍然能够将性能和价格很好地结合起来,并且也很容易实现彩色显示。

### 10.2.3 打印机

打印机是用于硬拷贝输出数据或文本的设备。通常分为击打式和非击打式两种类型,这取决于使用的打印机制的特性。击打式打印机使用机械打印机制,非击打式打印机主要依靠光学、喷墨或静电技术。

非击打式打印机只有很少的移动部件,因而能够高速运行。其中激光打印机使用的技术与影印机相同。用激光束扫描覆盖有带正电荷的光敏材料的转鼓,被激光束照射到的正电荷消失,然后带负电荷的色粉喷洒在转鼓上,这些粉末将粘附在正电荷上形成一页影像,紧接着转印到纸面上。转印完后将转鼓上多余的色粉清洗干净以备打印下一页。

另一种非击打式打印机是采用喷墨方式的,在这种方式中不同颜色的墨滴从细小的喷嘴喷射到纸页,从而产生彩色输出。用来喷洒墨滴的技术很多,例如,在气泡式喷墨打印机中,喷嘴与一个外加有热脉冲的小腔相连。热脉冲使小腔里的墨水蒸发形成气泡而将少量墨水从喷嘴挤出。当小腔里的气体冷却后又会产生真空而吸入一些新的墨水。喷墨打印机通常要比激光打印机昂贵,但打印出的图像质量更高。

大多数打印机形成字符和图形图像的方式与在视频屏幕上形成图像的方式相同,即在矩阵上打印点。这种方式很容易打印出各种字体,也可打印图形图像。但由于人眼对规则图案的敏感性,一个规则的点阵很容易被观察出来从而影响图像的视觉效果。高品质的打印机使用抖动技术来克服这个缺点。回想一下,我们知道每个像素由几个小点组成,每个小点的颜色为三原色中的一种。抖动技术就是使每一个像素中组成点的几何排列和颜色不断变化,这样就打破了规则图案的单一性,并且外观上具有更多的色彩。

最高品质的打印机通常用于图形艺术品和照片的打印。采用染料升华技术的喷墨打印机就适合这些应用。但它们也是最昂贵的。在这种打印机中通过控制墨水加热的温度来改变喷射在纸页上的墨水量,因此每个点的颜色深度可以连续改变。因为墨水在纸张上还会进行扩散,所以还需要使用特殊的纸张,以得到颜色的精确控制。

### 10.2.4 图形加速卡

在许多计算机应用中都包括高品质的图形图像。最常见的图形使用是在视频游戏中,其他应用还包括艺术作品、医学图像和动画电影等。高品质的图像需要大量的显示像素。在一个图像被发送到显示屏幕之前,必须计算出每个像素的颜色来并存储到一个存储缓冲区中。从那里,这些信息以至少30次/s的速度发送到屏幕以保持显示图像被刷新。

计算像素亮度和色彩的任务可以由软件来完成,结果图像可以存储在计算机主存的屏幕缓

缓冲区中, 从那里通过计算机总线将它发送给显示器。然而, 由于需要处理的数据量非常巨大, 以至于很容易就将处理器淹没, 而只剩下很少的计算能力给其他任务。而且, 使用计算机总线传送屏幕缓冲区的内容给显示器也会消耗相当一部分总线带宽。如果每个像素使用32位, 一个 $1024 \times 1024$ 像素的图像就需要用4MB的数据来表示, 这将导致存储总线上至少有120MB/s的传输量。

561

大部分图形应用都要求有显示三维(3D)物体功能。例如, 在计算机游戏中, 完全用软件创建一个完美视频图像的人工三维世界。产生这些图像的计算密度是非常高的, 最切实可行的方法是提供一个专门设计用来处理这些密集计算的处理器。这种处理器称为图形处理单元(GPU), 是安装在大部分个人计算机上很流行的图形卡的基本部件。图形卡还有一个大容量高速存储器, 一般在8MB到64MB的范围内。这个存储器在GPU进行计算时被使用, 同时它也存储将被传送到显示屏幕上的结果图像。图形卡是直接和显示器连接的, 这样刷新屏幕所需的数据传输就不需要使用计算机总线了。高性能的图形卡能够每秒钟刷新屏幕75到200次。

#### 图形端口

图形卡可以插入到计算机总线上, 比如PCI。但更常见的情况是, 在计算机主板上有一个称为图形加速端口(AGP)的特殊连接插槽, 图形卡插入到这个插槽里。AGP是一个32位的端口, 它能够支持的传输速率高于PCI总线。这种端口有AGP 1x、2x、4x和8x几种标准, 其中AGP 1x是最初的标准, 能提供264MB/s的数据传输速率。后来的标准能够支持的速率是这个速率的几倍, 其中AGP 8X能提供2GB/s的速率。

#### 图形处理

在计算机图形中, 一个三维物体是通过将其表面分割成大量的小多边形(通常是三角形)来表示的。在这个过程中, 第一项任务是将三维场景转换成二维表示, 这个二维表示应尽可能地与人眼观察到的图像相一致。首先通过投影和透视计算确定出三角形的顶点在二维图像中的位置, 这些三角形将用来表示场景中的各种物体。然后, 再用一个复杂的算法给每一个三角形确定一个适当的颜色和阴影, 由此形成一幅逼真的图像。这些计算需要考虑场景中的光源、各种表面的反射、阴影等因素。处理过程中很重要的一步是给表面增加一些纹理, 如木材纹路和砖墙的外观。纹理通常是用纹理像素制作出来。将一个纹理像素阵列施加在单个图像三角形上就可以产生出原始三维物体纹理表面的效果。场景中被隐藏了的部分在剪辑过程中删除掉以节约不必要的计算。处理的最后一步是取样, 在这一步中图像被取样来确定每一个像素的颜色和亮度。将整个3D场景分解为对像素(这些像素将被传送给显示器)的描述的过程称为渲染。

562

为了移动图像, 这些计算每秒钟必须重复很多次。要产生屏幕上平滑运动的外观, 图像的像素每秒钟至少需要重复计算20次, 一般是30到40次, 以得到高品质的视频图, 这称为帧频。视频卡必要的计算能力通常是用T&L(转换和照明)速率来衡量, T&L是指每秒钟的三角形数量, 对于这些三角形图形卡完成了投影、剪辑、照明和取样所需的全部计算。

举一个例子, 在表10-1中列出了ATI公司制造的RADEON VE图形卡的特性。*nVidia*公司制造的GeForce 2 MX图形处理器也能提供十分相似的功能。这两种图形卡在个人计算机中都非常流行。此外, 还有一些性能更高的专业图形卡。在这个计算机工业中飞速发展的领域, 功能更加强大的处理器不久就会出现。

表10-1 RADEON VE 图形卡

特 性	描 述
GPU芯片	RADEON VE
总线	AGP 4x
存储器	可达64MB,DDR SDRAM
色彩	32位,其中8位保留供以后使用
像素	2048 × 1536
T&L 速率	30M个三角形/s
屏幕刷新频率	75到200,图像分辨率越低,刷新频率越高
附加功能	支持TV、VCR、DVD、HDTV和压缩的MPEG2

### 图形软件

图形卡提供了很多复杂的特性,要使用这些特性需要专门为图形卡设计软件。在这个领域中只有非常少的标准,市场是完全开放的。仅仅在计算机上安装了一个更好的图形卡并不能自动地改善图形的质量,还需要有专门使用这个图形卡的软件。目前一些供图形软件使用的应用程序接口标准(API)已经开始出现。制定这些标准的目的是为了开发出独立于硬件的软件。以计算机游戏软件为例,有了这些标准它就可以在不同公司制造的图形卡下很好地工作,并使用每一个图形卡提供的特性。OpenGL(开放式图形库)就是这种标准的一个例子。逐渐地,图形卡将被设计成和OpenGL以及其他涉及图形处理的相似标准相兼容。

## 10.3 串行通信连接

键盘、鼠标等设备是直接和计算机连接起来的,一般是通过串行通信连接。其他设备如打印机和扫描仪,既可以和计算机连接,又可以通过通信网络与计算机连接,因此它们可以被几个用户共享。由于Internet在许多计算机应用中起着很重要的作用,所以计算机通常是永久的或通过拨号连接到Internet上。

**563** 在本章的剩余部分,我们主要讲述一些串行通信连接中常用的方式。首先讲述一些基本概念。

### 调制和解调

在数字电路中,我们用一个电信号表示一位,该电信号有两个电压值。如果在通信连接时使用相同的表示方法,那么认为该连接使用了基带。在另一种可供选择的方式中0和1是通过调制出正弦载波信号来表示的,这种方式也被广泛使用并称为宽带传输。例如,信号频率可以在两个不同值之间变换, $f_1$ 表示0, $f_2$ 表示1。在这种方式下,连接被认为使用了调频或移频键控(FSK)。此外,还使用了很多其他的调制方式,如可以改变载波信号的相位产生移相键控(PSK),或改变它的振幅产生调幅(AM)。正交调幅(QAM)是一种将载波信号的振幅调制和相位调制结合起来的方式。因为改变了两个参数,所以这种方式就有4种可能的组合,因此每个传输信号可以表示2位信息。

在任何时钟周期下传输的信号形态都称为码元(symbol)。因此,在FSK方式中就有两种可能的码元,分别由频率为 $f_1$ 和 $f_2$ 的正弦信号组成。在QAM方式中有四种可能的码元,分别由它们的振幅和相位定义。波特率是指每秒钟传输的码元的数量,即每秒钟信号状态改变的次数。只有在二元调制中,如FSK,波特率才与位速率相同。在QAM方式中,位速率是波特率的两倍,因为每一码元表示2位信息。还有一些调制方式使用了8、16或更多种码元。如在一个有16种码元的系统中,每个符号表示4位信息,因此位速率是波特率的4倍。

如图10-2所示,在通信连接的两端安装了调制解调器进行必要的信号转换。图中显示的是一台连接到网络服务器的计算机,计算机与网络服务器之间连接可以是永久性的连接,也可以是使用电话线的拨号连接。

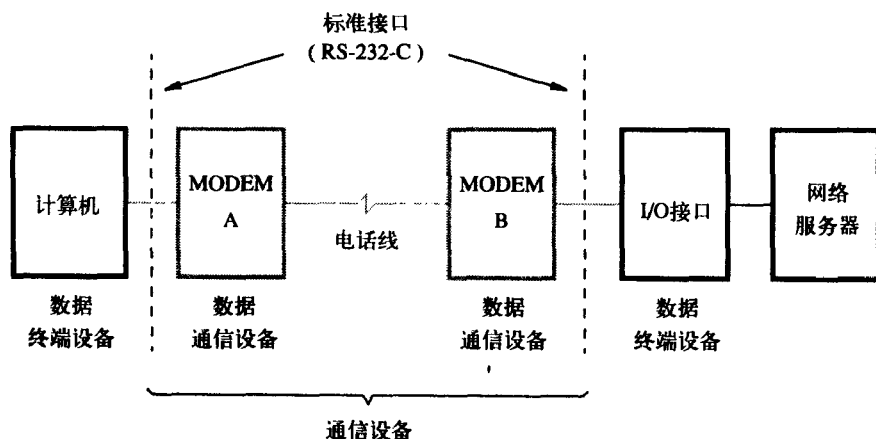


图10-2 远程连接到网络

### 同步

串行通信是指每次只发送一位数据。这要求发送和接收设备都要使用相同的定时信息来解释每一位信息。当通信设备彼此对对方物理性关闭并且有多条信号线路可用时,时钟信号就可以与数据一起传送过去。然而,对于更长一些而且只有单条信号线路可用的连接就不可行了。更重要的是,即使有第二条线路可用,数据信号和时钟信号遇到的延迟可能也是不同的。因为这些原因,定时信息与数据就被编码在同一个传输通道中。很多的编码方式已经被开发出来了,这些编码方式使接收器能够正确解码收到的信号并复原定时信息和传输的数据。

实现串行传输有两种基本的方法。对于速度不超过几十kb/s的数据传输,可以采用一种简单的方式。在这种方式中发送器和接收器使用独立的时钟信号,这两种时钟信号有相同的额定频率,但不需要保证两个时钟信号有完全相同的相位和频率。因此这种方式被称为异步传输。后面将要简要介绍的起止方式就是这种方法的一个普通例子。

在高速传输数据时需要使用同步传输。在同步传输中,接收端连续地监测收到信号的跳变位置并相应调节本地时钟的相位,以此来复位发送器的时钟定时信息。这样接收端的时钟就与发送器的时钟同步了,可以用来正确复原传输的数据。可以用来在同步连接上编码定时信息的技术很多,由于利用带宽的能力不同从而得到不同的数据传输速率。

### 全双工和半双工连接

一个通信连接可以按以下三种方式之一进行操作:

单工方式,只允许在一个方向上进行传输;

半双工方式(HDX),允许在两个方向上进行传输但不能同时进行;

全双工方式(FDX),允许同时在两个方向上传输。

单工方式只适用于在远程位置上只有输入和输出设备两者之一的情形。因此,单工方式很少用到。而半双工和全双工的选择要综合权衡经济和操作速度两个因素。

使用最简单的电路装置,一对线路可以使传输能够并且只能在一个方向上进行,这就是单工操作。而要实现半双工连接,在两端必须使用开关将接收器或发送器连接到线路上,但开关



不能同时连接接收器和发送器。当一个方向上的传输完成后,开关逆置使得相反方向上的传输可以进行。开关的方向由线路两端的设备控制。

全双工操作可以在四线连接上实现,每个传输方向占用两条传输线。也可以通过使用两个互不重叠频带在两线连接上实现。这两个互不重叠频带可以产生两个独立的传输通道,每个方向上各一个。另外,全双工操作还可以在线路的两端使用混合器的普通频带上实现。混合器的作用是将传输方向相反的信号分开使它们互不干扰。拨号电话连接使用的就是这种类型的线路。

在同步半双工操作中,当传输方向逆置时会产生时间延迟,因为发送器的调制解调器必须发送一个初始化信号序列,使接收器调节通道环境。延迟的长短取决于调制解调器和传输设备,可能从几毫秒到一百多毫秒不等。

上面讲述的是有关传输连接和调制解调器的特性。此外,数据传输的本身特性和系统对传输错误的反应方式也是影响选择半双工或全双工的重要因素。在这里我们只讨论第一个因素。

许多计算机应用要求计算机接收输入数据,进行一些处理,然后返回输出数据。对于这类应用,半双工连接就能满足要求。但是,如果线路两端交换的信息短且频繁,转换传输方向引起的延迟就不可忽略了。因为这个原因,实际中许多应用都使用全双工传输方式,即使实际的数据传输从不同时在两个方向上进行。

在有些情况下,在两个方向上能够同时传输数据有着相当大的优势。如图10-2所示的系统中,计算机的使用者可能希望将计算机作为一个视频终端直接与网络服务器对话。如果这样的话,在键盘上输入的每一个字符都应该回显在计算机屏幕上,这项功能可以由计算机本地完成,也可以由网络服务器远程完成。使用远程回显时还可以提供一个自动检查的功能来确保传输过程中没有发生任何错误。使用远程回显时,如果采用半双工连接,那么下一个字符必须延迟到前面的字符回显后才能发送。然而如果采用全双工操作就没有这种限制了。全双工传输适用的另一个例子是高速计算机通信网络中节点间的连接。在任一个连接中相反方向上传输的信息之间是没有任何联系的,因此它们就可以同时传输。

### 10.3.1 异步传输

串行通信最简单的方式是采用起止技术的异步通信。为了便于复位定时信息,数据被组织成带有明确定义的起点和终点的6到8位的小组。典型情况下,字符在8位内编码按照图10-3所示进行传输。连接发送器和接收器的线路闲置时状态为1,传输字符前有一位0作为起始位,后面紧接着的是8个数据位和1或2个终止位,终止位的逻辑值为1。起始位的作用是提醒接收器数据传输即将开始,它的前沿用来同步接收器和发送器的时钟。末端的终止位在连续传输时用来描述连续的字符。当传输结束时,终止位之后线路保持1状态。插入和删除起始位和终止位由发送器和接收器电路完成。

为了保证接收器能正确同步,接收器的时钟取自于一个频率远高于传输速率的本地时钟,一般要高出16倍。这就是说在每一个数据位间隔期间可以产生16个时钟脉冲。这个时钟用来使一个模16的计数器增大,该计数器当检测到起始位的前沿时被重置为0。当计数达到8时,表示已经达到了起始位的中间位置。这时,起始位的值被取样来确认它是一个有效的起始位,同时计数器再一次被重置为0。从这一点往后,每当计数达到16就对输入的数据信号取样一次,此时应当接近每一传输位的中间位置。因此,只要传输字符中位的相对位置发生错误的长度不超过时钟周期的一半,接收器都可以正确解释字符编码中的每一位。

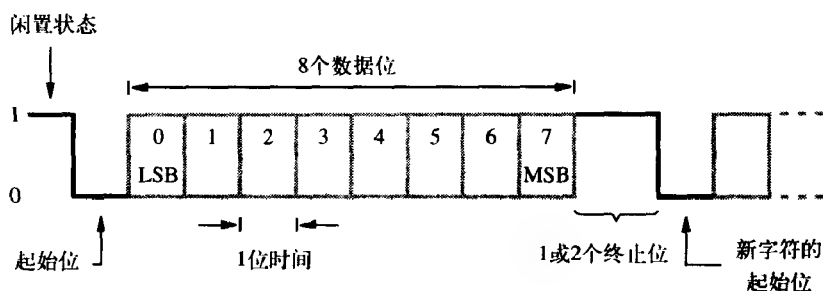


图10-3 异步串行字符传输

在商业性的设备中有许多传输速率标准，从300到56 000比特/秒的范围内。起止传输方式主要用于短距离的连接，如图10-2中计算机和调制解调器的连接。对于更长距离的连接，如图中两个调制解调器之间的连接，起止方式只能在很低的速度下使用。高速调制解调器一般使用下一节将要讲述的同步传输方式。

在传输字符时，字符通常用7位ASCII码（见附录E）来表示，占用图10-3中的位0到位6。MSB为传输字节中的第7位，通常置0。此外，它也可以用作奇偶校验位以辅助检测传输错误。奇偶校验位的值等于位组的总和 $\text{mod } 2$ 。因此，如果传输的数据包含奇数个1，那么它就等于1，否则就等于0。使用奇偶校验位时它是由发送器设置的，这样传输的8位数据的奇偶性就不会改变了，或奇或偶。如果传输错误导致其中一位的值被改变，接收器就会检测到，从而可以确定已经发生了错误。

567

ASCII码字符包括字母、数字和特殊符号（如\$、+和>），也包括很多非印刷字符，例如，EOT（传输结束）和CR（回车）。这些字符可以用来请求特殊的操作，尤其是在给远程计算机发送信息或从远程计算机上接收信息时。

### 10.3.2 同步传输

上面描述的起止方式中，起始位的开始有一个1到0跳变，如图10-3所示，这个跳变的位置是获取正确定时信息的关键。因此，只有在传输速度足够低和传输连接中的方形波形状保持不变的情况下，这种方式才是可用的。对于高速、长距离的连接常常会发生大量的信号衰减。图10-4中显示的许多位在彼此的顶部重合的情形，说明了波形是怎样从一个位的位置变化到另一个位的位置的。信号衰减是由信号变形之类的因素引起的，这些因素是由线路、传输设备、邻近干扰源、抖动（信号跳变位置的随机变化）等导致的。根据每一位中间区域的形状，该图称传输连接的眼孔图。复杂的编码和解码方法被用来帮助接收器确定眼孔图的中点位置，在这个中点位置1和0相隔最远，这是对接收到的信号进行取样的最佳点。

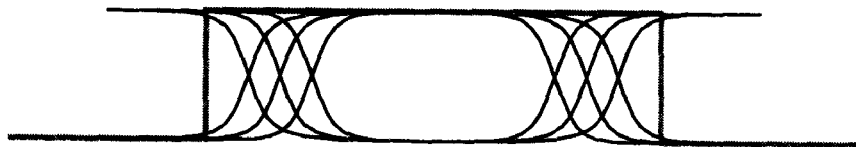


图10-4 连续位的重叠跳变眼孔图

在同步传输中，数据是以块为单位进行传输的，每一块包括几百到几千个比特。在每一块的开始和结束都使用了适当的代码来标记，块内数据是按照约定的规则组织的。调制解调器需

要一段较长的起止时间来完成发送和检测载波频率、建立同步等操作。在一些调制解调器中,起止时间也被用来调节调制解调器电路与连接的传输特性相适应。

### 网络连接——ADSL

近年来,特别是随着网络的广泛使用,在家里或办公室通过高速连接接入Internet的需求越来越多。到目前为止,现有的调制解调器仍不能满足需求。传统的调制解调器将数字信号转换为模拟形式,使用电话线的4kHz的声音频带。当一台计算机使用这种调制解调器与另一台计算机通信时,这条线路就不能打普通电话。更为重要的是,传输速率只能达到几十kb/s,这远小于连接到远程服务器或Internet所需的速度。

传统的电话技术只利用了电话线的很小一部分信息传输能力。依赖于传输距离和线路状况,现代通信技术可以使电话系统中的双绞线上的最高速度达到50Mb/s。许多方案已经被开发出来,利用这些尚未使用的传输能力,这些方案直接以数字形式在用户家中或工作中与电信公司的电话总机之间传输信息。电话公司将电话总机与用户之间的连接称为用户环路。因此,当采用数字形式传输时,这种方式被称为数字式用户环路(DSL)。

计算机通信完全依赖于不同部门提供的设备和服务之间的协同工作,这些部门包括计算机公司、调制解调器制造商、网络服务提供商。因此,在一些被大家都接受了的标准上达成一致是十分必要的。在DSL领域,只有很少的标准,包括SDSL(对称DSL)、HDSL(高速DSL)和ADSL(不对称DSL)。其中,ADSL是将家用个人电脑连接到Internet上使用的最广泛方式。下面我们简要讲述一下这种方式的主要特性。

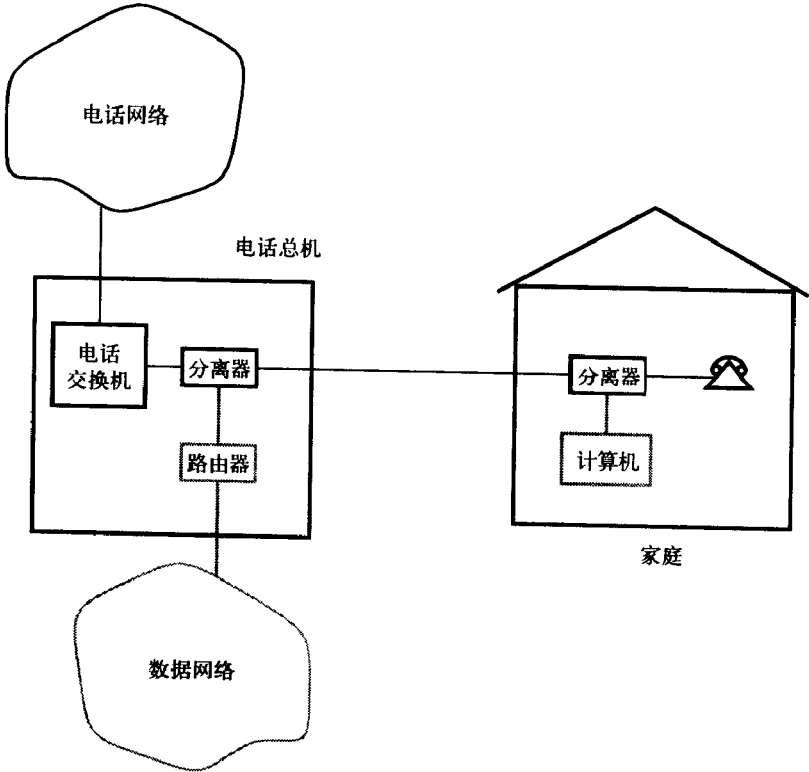
ADSL中的不对称是指上行方向和下行方向的传输速率不同。在大部分时间里,从计算机发送到Internet服务器(上行方向)的信息是用户的输入信息,这时只需一个低速连接就足够了。而流向用户的信息,如在计算机屏幕上显示图像,则需要高速连接以提供良好的响应。因此,ADSL中下行方向的传输速度要比上行方向的传输速度大得多。

ADSL方式使用不同的频带和时分多路复用技术得到多个通信通道。其中一个分配给日常的电话,其他的分配给上行方向和下行方向上的数据传输。图10-5所示的是一种典型的布局。图中使用单根双绞线在电话总机与用户之间传输信息,在每一端有一个分离器将数据传输与声音信号分开。在用户端,数据通过适当的数据连接如以太网和USB连接输入到计算机中,声音信号被发送到电话机。在电话总机端,数据被发送到与Internet相连的路由器,声音信号被发送到电话交换机。(路由器是一个用来在数据网络中指挥传输的交换设备。)采用这种布局可以使计算机不拨号就能与Internet随时保持连接,同时日常的拨号电话服务也可以继续使用。

### 电缆调制解调器

电缆调制解调器是另一种将家庭计算机连接到Internet的方法。它使用有线电视连接取代了电话连接。由于有线电视中使用的同轴电缆的带宽要比双绞线大得多,因此使用电缆调制解调器能达到的最高速度要高于DSL的速度。然而,由于有线电视对一个邻近区域的所有用户使用的是类似于总线的连接,所以电缆的信息运输能力被所有连接用户共享。对于每一个用户,全部运输能力只有在这根电缆上的其他用户都未使用时才能达到。典型的电缆调制解调器布局如图10-6所示。

电缆调制解调器系统中单个用户获得的最高速度依赖于网络服务的提供商,可能从600kb/s到10Mb/s不等。



570

图10-5 ADSL连接

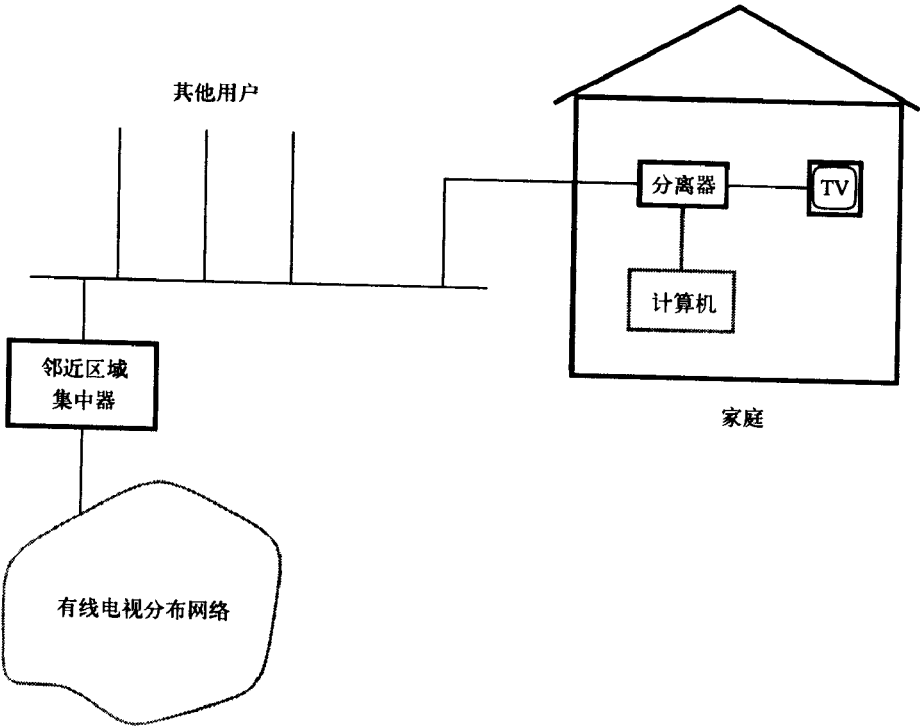


图10-6 电缆调制解调器连接

### 10.3.3 标准通信接口

标准接口指的是将两个设备连接起来的点的集合。其中一个已经得到广泛认可的标准是EIA（电子工业协会）的RS-232-C。在北美以外的地区称之为CCITT Recommendation V24。这个标准完整地规定了数据通信设备之间的接口，如调制解调器与数据终端设备（如计算机）之间的接口。RS-232-C接口由25个连接点组成，表10-2对它们进行了描述。

下面看一个简单又常见的例子。考虑图10-2中的连接，假设使用的是电话拨号网络。所用的调制解调器可以接通和挂断、发送拨号音和检测收到的振铃信号。采用的传输方式是前面所描述的FSK方式，能进行全双工通信。拥有两个传输信道，每个方向一个，一个信道使用频率1275Hz和1075Hz，另一个使用频率2225Hz和2025Hz来分别表示逻辑1和0。

图10-7给出了建立连接、传输数据和终止连接所需的逻辑信号序列。该过程包括的步骤简要描述如下：

- 571
1. 当网络服务器准备好接收一次访问时，将数据终端就绪信号（CD）置1。
  2. 调制解调器B监控电话线，当检测到一个振铃电流时，表示收到一个呼叫，就将振铃指示器（CE）置1通知服务器。如果当振铃电流被检测到时CD=1，调制解调器就自动应答呼叫，然后将调制解调器的就绪信号（CC）置为1。
  3. 服务器通过将请求发送（CA）置1指示调制解调器B开始发送代表1的频率（2225Hz）信号。完成这个操作后，调制解调器B将允许发送（CB）置为1。当调制解调器A检测到这个频率的信号后，就将接收线路信号检测器（CF）置为1。
  4. 计算机将CA置为1。调制解调器A发送1275Hz的信号并将CB和CC置为1。当调制解调器B检测到1275Hz频率的信号后将CF置为1。
  5. 至此一个全双工连接就在服务器与计算机之间建立了，它可以在两个方向上传输数据。接口引脚BA（发送数据）和BB（接收数据）数据传输，接口中所有其他信号保持不变。
  6. 当使用者要结束时，服务器将请求发送和数据终端就绪信号CA和CD置0，使调制解调器B终止2225Hz的信号并从线路上断开。信号CB、CF和CC也被调制解调器B置0。当调制解调器A检测到线路上的信号消失时，将接收线路信号检测器（CF）置0。
  7. 调制解调器A将它的1275Hz的信号从线路上撤销，将CB和CC置为0，并挂断。
  8. 服务器将数据终端就绪（CD）置1，准备下一次呼叫。

表10-2 EIA标准RS-232-C信号一览表（CCITT Recommendation V24）

名称		引脚号	功能
EIA	CCITT		
AA	101	1	保护地
AB	102	7	公共地返回信号
BA	103	2	发送数据
BB	104	3	接收数据
CA	105	4	请求发送
CB	106	5	允许发送
CC	107	6	数据通信设备就绪
CD	108.2	20	数据终端就绪
CE	125	22	振铃指示器

(续)

名称		引脚*号	功能
EIA	CCITT		
CF	109	8	接收线路信号检测器
CG	110	21	信号质量检测
CH	111	23 S	数据信号速率选择 (从DTE+到DCE‡)
CI	112	23 S	数据信号速率选择 (从DCE‡到DTE+)
DA	113	24	发送信号单元定时 (DTE+)
DB	114	15	发送信号单元定时 (DCE‡)
DD	115	17	接收信号单元定时 (DCE‡)
SBA	118	14	二级发送数据
SBB	119	16	二级接收数据
SCA	120	19	二级请求发送
SCB	121	13	二级允许发送
SCF	122	12	二级接收线路信号检测器

\* 引脚9和10用于测试, 引脚11、18和25闲置  
+ 数据终端设备  
‡ 数据通信设备  
S 该引脚上信号的名称取决于信号的方向

572  
573

使用调制解调器进行初始连接的过程包括信息交换, 在交换的信息中, 双方在将要使用的编码方式、传输速度、数据块大小等参数上达成一致。RS-232-C接口能够为两个数字化设备提供串行连接。单个信号如CA和CD的解释依赖于具体设备的功能。如果不需要这些信号, 双方设备就将它们忽略掉。大部分应用中使用的信号不超过表10-2中所列的9个信号。

10.4 结束语

本章主要介绍了输入和输出设备的概况以及它们的操作原理。I/O设备是计算机系统的基本组成部分, 它们为向计算机输入信息和从计算机接收结果建立了连接。近年来又出现了许多创新的设备。高品质且价格可以接受的输出设备已经向个人计算机提供了。可用的输入设备的范围也在继续扩大, 包括数字照相机和各种手持设备。

本章也介绍了计算机通信方面的内容, 特别是在串行连接中使用的基本技术。串行连接普遍应用于连接计算机与I/O设备或其他计算机。还介绍了一些通过普通传输设施(电话网络)和有线电视网高速连接到Internet的例子。这些设备的广泛使用产生的互连性, 改变了我们使用计算机的方法并开发出了大量的家庭和商业上的应用。正是计算机和通信两个领域的合作引领了当今的信息技术时代。

574

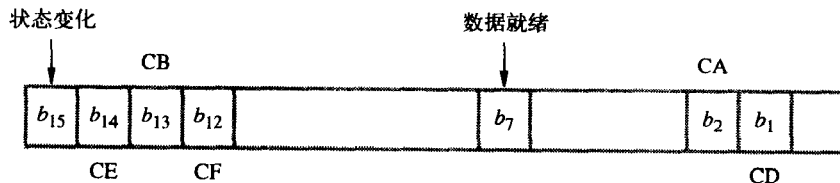
习题

10.1 视频显示器每秒钟必须至少刷新30次以保持屏幕不闪烁。在每一次完全扫描屏幕时, 用来照亮每个点的总时间是1 $\mu$ s。然后电子束熄灭并移至下一个要被照亮的点。平均起来, 将电子束从一点移动到另一点需要花费3 $\mu$ s。由于受能量损耗的限制, 电子束的照亮时间不能超过这段时间的10%。计算屏幕上能被照亮的点的最大值。

步骤	计算机	接口信号	调制解调器A	调制解调器B	接口信号	服务器
1				允许自动应答	CD	← 1
2	拨号数字 →			1 → 摘机 1 →	CE CC	
3		CF	← 1	← 2225 Hz 1 →	CA CB	← 1
4	1 →	CA CB CC	1275 Hz → ← 1 ← 1	1 →	CF	
5	输出数据 ← 输入数据 →	BB BA	← 数据 1275/1075 Hz →	← 2225/2025 Hz 数据 →	BA BB	← 输出数据 → 输入数据
6		CF	← 0	终止2225Hz并断开 0 → 0 → 0 →	CA CD CF CC CB	← 0 ← 0
7	(0 →)  终止连接	CA CB CC	撤销1275Hz ← 0 ← 0			
8					CD	← 1

图10-7 RS-232-C 标准信号发送序列

- 10.2 有一个通信信道,它使用8进制信号代替二元通道中使用的2进制信号。如果这个信道的波特率为9600波特/秒,那么它每秒钟传输多少比特?
- 10.3 有以下组件可用:
- 一个6位二进制计数器,有时钟、清零输入和6个输出
  - 一个3位串行输入并行输出移位寄存器
  - 一个运行速率是输入速率8倍的时钟
  - 有预置和清零控制的逻辑门和D触发器
- 使用这些组件设计一个电路将3位串行数据从数据输入线路装入到移位寄存器中。假设数据的格式如图10-3所示,但只有3位而不是8位数据。你设计的电路应该有两个输出A和B,两者初始时都被清0。如果在数据位后检测到一个终止位,就将输出A置1。否则,输出B置1。对你的设计给出操作说明。
- 10.4 一个两台计算机之间的异步连接使用起止方式,有一个起始位和一个终止位,传输速率为38.8Kb/s。这两台计算机可得到的有效传输速率是多少?
- 10.5 一个通信连接对每个传输字符使用了奇校验。参考附录E,给出字符A、P、= 和5的8位传输格式。
- 10.6 有一个通信线路调制解调器通过RS-232-C接口连接到计算机上。计算机可以通过一个16位寄存器访问这个接口的控制信号,如图P10-1所示。当位 $b_{12}$ 或 $b_{13}$ 的状态发生改变或 $b_{14}$ 被置1时,状态变化位 $b_{15}$ 被置1。当寄存器被处理器访问后,位 $b_{15}$ 被清0。根据图10-7中的步骤1到步骤4,为第3章中的一个处理器写一个程序来执行建立电话连接所需的控制序列。



图P10-1 习题10.6中调制解调器接口的I/O寄存器结构





### 本章目标

在本章中你将学习以下内容:

- 实现RISC ISA的ARM、PowerPC、Sun SPARC、Compaq Alpha以及Intel IA-64处理器系列
- 实现CISC ISA的Motorola 680X0/ColdFire和 Intel IA-32处理器系列
- 具有操作堆栈数据结构指令的Hewlett-Packard HP3000

577

第2章介绍了如何在汇编语言级别上实现编程的基本概念,并说明了各种必要的机器指令和寻址方式。第3章中我们采用ARM、Motorola 68000和 Intel IA-32指令集的体系结构作为例子。在本章中将继续讨论这三个指令集体系结构 (ISA),并详细说明实现ISA且在一定价格和性能区间的处理器系列中各个成员的特征。ARM体系结构具有RISC的特征, Motorola 和Intel IA-32体系结构具有CISC的特征。我们要描述的RISC型PowerPC体系结构,它的处理器和IA-32处理器具有完全相同的市场定位。还要讨论具有64位地址和数据宽度的Sun SPARC、Compaq Alpha和Intel IA-64 RISC体系结构。这些64位处理器主要用于高性能的工作站和服务中。最后,我们会用一种截然不同的基于历史重要性考虑且在Hewlett-Packard HP3000中使用的方法,来展示如何为一台机器组织一套指令集,使它通过使用堆栈数据结构来保存操作数,从而大大简化它的计算。

第2章、第3章、第7章和第8章中描述了采用RISC和CISC方法进行处理器设计。在举例说明它们的特征之前,我们简要回顾一下每种方法的显著特点。CISC指令集提供了很多功能强大的指令,这些指令为高级语言操作和程序排序控制结构提供更为直接的实现方式。然而,这些指令的执行相对来说也更加复杂。CISC的合理之处在于对于一个给定的高级语言程序,它会使用较少的机器语言指令,这样可以减少程序的执行时间。如果复杂指令能够快速高效地执行,那么的确可以实现减少程序的执行时间。实际上,实现具有高执行效率的指令已经被实践证明是极具挑战性的,并且要求具有比较大的芯片面积。此外,CISC指令集很难在优化的编译器上使用。

因为RISC指令必须执行一些给定的计算任务,使用相对简单的RISC方法最初与CISC相比效率较低。然而,RISC指令非常适合用于可以提高执行效率的流水线操作。RISC指令集最重要的优点是它可以有效使用在优化编译器上。另外一个优点是关于超大规模集成电路 (VLSI) 制造技术。由于在RISC处理器中指令操作和顺序控制所需的芯片面积更小,这样可以有更多的空间来设置更多的寄存器和片上高速缓存。

种种因素表明,尽管应用两种方法都生产出了十分具有竞争实力的商业产品,但是,从20

世纪90年代开始开发的新机型中RISC型ISA已经具有了明显的优势。

578 正如前面几章中阐述的那样，我们必须注意在设计能高速执行指令的处理器时，除了指令集的类型外，还要考虑很多重要的因素。正如在第8章中描述的，多道、流水线和能够执行超标量程序的功能单元都是必不可少的。另外，从高级语言程序生成高效率机器语言代码的优化编译器也必须随着硬件的发展而不断发展。

## 11.1 ARM 系列

在第3章的部分I中，我们以ARM指令集体系结构作为RISC类型的例子来描述指令集的设计。ARM处理器主要用于嵌入式系统的应用中。因此，大部分的实现要求必须是低成本和低功耗的。很多情况下，如移动电话，目标设备是电池供电的，使用的电压范围是1到3伏。相对于面向PC市场的高性能Intel Pentium处理器而言，低档的ARM处理器降低复杂性就有更大的意义了，它仅需要Pentium芯片中的部分晶体管电路。我们将通过讨论ARM ISA的不同实现方法来阐述这些问题。Furber<sup>[1]</sup>的著作中对ARM历史、体系结构和实现作了大量的阐述。《IEEE Micro》<sup>[2]</sup>中的文章和ARM网站<sup>[3]</sup>也提供了详细的信息。

从20世纪80年代中期到2000年，ARM ISA已经具有了5个主要版本，称为v1到v5。版本v3在第3章中已经描述过了，这个版本采用ARM7处理器，于90年代中期开发实现的。后续章节将描述处理器的实现机制及其特点。

相对于版本v3，我们也描述一下其他版本的一些特点。版本v1和版本v2仅支持26位存储地址，从v1到v2增加了32位的乘法指令。版本v3全部采用的32位字节寻址和32位的字操作数。版本v4中包含64位和32位乘积的乘法指令，又增加了Load和Store指令，其操作数是16位（半字）数据。版本v5以及标注为v5E的扩展版本增加了专有指令：管理程序调试时的软件断点，为软件进行浮点运算而进行的规格化，以及在进行16位操作数的加法和乘法操作时允许快速处理数字符号的程序。

除ARM ISA的这5个版本以外，还有一种将指令编码成32位字格式，并与版本v4和v5同时存在的一种压缩的编码子集。我们将在后续章节中进行阐述。

### 11.1.1 Thumb指令集

ARM ISA说明书中包含全集指令版本v4和v5的一个压缩编码子集。该子集称为Thumb指令集，并且版本名扩展成v4T和v5T，表示包含的意义。所有的Thumb指令都编码成16位的半字格式。

579 Thumb指令的实际作用是它可以使程序占用存储空间减小，以便可以用在低成本、低功耗的嵌入式系统应用中。ARM7TDM1处理器实现了体系结构中的v4T版本。该处理器连同一个小容量的RAM存储器和数字信号处理硬件一起设计在同一块芯片上，非常适合应用于移动电话。

由Thumb指令编制的程序按照如下方式执行。指令从存储器中读取并动态地（也就是，在执行时）从高度编码的16位格式“解压缩”成相应的标准32位ARM指令，然后再执行。这就是该指令是如何在主流的低端处理器中执行的方式。在一些高性能的处理器中，Thumb指令可以省略解压缩成标准的ARM指令的步骤，直接解码成可执行码。在当前程序状态寄存器（CPSR）中有一位标志T的状态位，判断输入指令流是采用Thumb指令（T=1），还是标准的32位ARM指令（T=0）。应用程序可以包含Thumb指令和标准指令的混合形式。

Thumb指令和标准指令之间的两个主要区别是：Thumb指令采用双操作数格式，目标寄存器

的号码是源操作数寄存器中的一个；而在标准的ARM指令中使用条件执行指令，在Thumb指令集中主要用转移指令。这些变化会明显节省指令编码位的空间。

### 11.1.2 处理器和CPU内核

ARM公司专门生产ARM处理器和与之紧密相关的部件，比如高速缓存和存储器管理单元。生产嵌入式系统和其他特殊应用的计算机产品的厂家需要把这些设计增加到他们的产品中。在大多数情况下，ARM的设计是和特殊应用硬件集成在同一块芯片上的。为了达到所需的用途，ARM设计被称为核。ARM提供两种不同形式的设计：硬件宏单元或综合形式。硬件宏单元是针对特定芯片制作过程所给出的一个详细物理设计图。综合形式是高级语言软件模块，该模块可以在需要的目标技术中使用一个合适的单元库进行合成。这种形式基于处理器的功能可供选择，并且很容易增加和删除。ARM7TDMI处理器是一种硬件宏单元内核，ARM7TDMI-S则是它的综合版本。

ARM设计可以用处理器内核或CPU内核进行分类。处理器内核仅仅包含一个处理器及与之相连的地址和数据总线。CPU内核除了包含处理器以外，还包含高速缓存和存储器管理部件。这里CPU这个名字有些不准确，因为传统意义上它是指中央处理单元，但是在此处使用是因为它是一个可识别的ARM术语。现在，我们简要描述一些具有代表性的处理器和CPU内核。

580

#### ARM7TDMI处理器内核

这个内核通常用在低成本、低功耗的应用中。处理器有一个简单的3段流水线，即由读指令、编码和执行3个阶段构成。它实现了版本v4T的体系结构，支持Thumb指令集和标准指令集。典型的运行参数是3.3V电压和66MHz的时钟频率。另外这个内核还能支持运行电压0.9V，为低功耗的电池供电应用进行综合，也能为达到更高性能而使时钟频率超过100MHz的应用进行综合。

#### ARM9TDMI 和 ARM10TDMI 处理器内核

这两个处理器内核分别采用5段和6段流水线。相对于ARM7TDMI的性能，它们有单独的指令和数据端口，可以提供更高的性能。当处理器的时钟频率为200MHz和300MHz时，该处理器内核中的版本7、版本9和版本10的性能级别比是1:2:4。相对其他两种处理器的32位通道宽度，ARM10TDMI的每个存储器端口通道宽度是64位。ARM9TDMI 和 ARM10TDMI分别实现了ISA的版本v4T和 v5TE。它们都是直接对Thumb指令进行解码执行的。在这两种内核中为了达到更高的性能级别都采用了高速缓存进行存储。

#### ARM720T CPU内核

该内核是由ARM7TDMI处理器内核、一个8K字节统一的指令数据缓存以及虚拟存储器管理硬件构成。缓存结构是16字节的块，并且采用4路组相联装置。存储器管理单元使用一个64入口的连接转换监视缓冲区，该缓冲区中存放最近一次的转换。这种集成单元的时钟频率可以达到60MHz。附加的缓存和MMU电路使得芯片的面积增加为单独处理器所需面积的5倍，功耗是原来的3倍。

#### ARM920T和ARM1020E CPU内核

这些CPU内核以ARM9TDMI和ARM10TDMI处理器内核为基础，并具有分离的指令缓存和数据缓存。ARM920T中的每个缓存包含16K字节，有32字节块，有64路的组相联装置。每个存储器端口都有一个MMU，并且每个MMU都有64入口相连的TLB。ARM1020E每个缓存有32K；此外，缓存和MMU与前者相同。

### StrongARM SA-110 CPU 内核

[581]

StrongARM CPU内核是ARM公司与DEC公司(后并入康柏公司)合作开发的,SA-110版本是Intel公司制造的。处理器部件实现了版本v4的体系结构。它不支持Thumb指令集。否则,该处理器可以与ARM9TDMI处理器内核相媲美。StrongARM SA-110的性能与ARM920T相似,但是它是采用早期技术实现的,并且在200MHz时钟频率下的功耗较大。

StrongARM 处理器有5段流水线,有分离的16K的指令和数据缓存。每个缓存有32字节块和32路组相联装置,每个转换监视缓冲区拥有32个入口。为了在数字信号处理应用中具有良好性能,所设计的高速乘法器电路会产生3个时钟周期以上的延迟。

## 11.2 Motorola 680X0 和 ColdFire系列

在第3章的部分II中我们介绍了Motorola 68000处理器。这里讨论680X0系列中的相继处理器的主要特征,以及与之有密切关系的ColdFire系列。在Tabak<sup>[4]</sup>的著作中和Motorola网站<sup>[5]</sup>上对这些处理器都有相关的描述。下面首先对Motorola处理器进行一般性的说明。

68000处理器于1979年诞生。经过20世纪80年代和90年代初,相继问世的68000、68020、68030和68040的主要目标是个人计算机市场,这些处理器由苹果计算机所采用。最新的680X0系列成员是90年代中期问世的68060。与68060关系最为密切的是ColdFire系列,该系列处理器主要面向嵌入式系统市场。

### 11.2.1 68020处理器

由于68020在体系结构上进行了重要改进,因此其功能比68000强大许多。改进的VLSI技术和更大的外包装避开了由于引脚的局限性带来的诸多约束,从而为其进一步的改进创造了可能性。这里关于68020的讨论也适用于68030和68040。稍后,我们再描述68030和68040性能中的其他改善。

[582]

68020提供32位地址和32位数据的外部连接。尽管数据总线宽度是32位的,它每次可以有效地处理设备之间8位、16位和32位的数据传输。该处理器可以根据特定设备的需要动态调整数据总线的宽度,而这种方法对于程序员来讲是透明的。68020总线包含有控制线,这些控制线由相连接的设备激活,指示该设备进行数据传输时所需要的总线宽度。这样,处理器就能处理不同宽度的数据传输,而无需在数据传输初始化之前知道实际的传输宽度。

68000中规定字操作必须从偶地址开始进行,在68020中取消了这条限制,即任意长度的操作数可以在任意地址开始。这就意味着16位和32位的操作数在主存中可以占用两个相邻的32位位置。因此,读取操作数需要两个访问周期,这样会对性能有一定的影响。这两步访问是由处理器自动执行的。处理器可以从地址中判断必须访问哪个32位的位置,以及这些位置的每个字节采用什么样的组装方式才能获得想要的操作数。

#### 寄存器集和数据类型

与68000一样,68020中的操作分为用户模式和管态模式。在用户模式下,寄存器变量必须与图3-18中给出的相同。而在管态模式下,68020有几个额外的控制寄存器来简化操作系统软件的执行。

68000可访问的数据单元是位、字节、字、长字和压缩的BCD码。除此以外,68020还可以处理四倍长字、非压缩的BCD码和位字段的数据类型。四倍长字由64位组成,非压缩的BCD每

个字节是一个BCD数。位字段由32位长字中的一位变量构成，它通过最左端的位及字段中的位数来指定。

### 寻址方式

表3-2中给出的所有68000的寻址方式，在68020中也都是有效的。为了增加对数据和地址表结构访问的灵活性和高效性，在68020中又增加了几种变址方式的版本。

全变址方式是比较强大的方式，因为它允许一定范围的位移量或偏移量，并提供一个比例因子。回想一下68000的全变址方式的语法：

$$\text{disp} (An, Rk, \text{size})$$

其中，位移量是一个带符号的8位数，指定的size用来表示寄存器Rk是32位的还是16位的，并用来计算有效地址。在68020中这种方式的指令允许位移量的值是8位、16位或32位。Rk中的内容进行乘法运算时引入一个比例因子。该比例因子的值可以是1、2、4或8。该方式的语法是：

$$(\text{disp}, An, Rk, \text{size} * \text{scale})$$

注意位移量在括弧中给出。有效地址EA的计算如下：

$$EA = \text{disp} + [An] + ([Rk] \times \text{scale})$$

当处理的表项长度是1、2、4或8个字节时，这种方式非常有效。如果适当选择比例因子使之等于表项的长度，那么表中的表项就可以按照对寄存器Rk内容每次增1的方式进行连续的访问。

另一种变址寻址方式的强大扩展版本是存储器间接变址方式，该方式中的操作数地址是直接读取主存的地址。这两种方式共同存在。在存储器间接传递变址（memory indirect postindexed）方式中，进行正常的变址处理之前，先取出存储器地址。其语法是：

$$([\text{basedisp}, An], Rk, \text{size} * \text{scale}, \text{outdisp})$$

583

有效地址计算如下：

$$EA = [\text{basedisp} + [An]] + ([Rk] \times \text{scale}) + \text{outdisp}$$

注意使用的两个位移量。16位或32位的基本位移量用来修改An中的地址，然后用来从存储器中取出操作数的地址。这样就可以从存储器中保存的地址表中选择一个地址，该地址表的起始地址在An中给出。第二个位移量是用于变址寻址的一般位移量，为了与基本位移量区别，它被称为外部位移量。

第二个版本是存储器间接预变址（memory indirect preindexed）方式，其中大部分的变址修改是在取出地址操作数之前完成的。该方式的语法是：

$$([\text{basedisp}, An, Rk, \text{size} * \text{scale}], \text{outdisp})$$

有效地址的计算如下：

$$EA = [\text{basedisp} + [An] + ([Rk] \times \text{scale})] + \text{outdisp}$$

在这两种方式中，An、Rk、basedisp和outdisp的值是可选的，除非用户特殊指定，否则这些值在计算有效地址时不被使用。当处理连续存储的列表时，存储器中存储的是数据项地址而不是数据项本身时，这种寻址方式是非常有用的。数据项可以存储在任何位置上。

所有变址方式的相关版本中程序计数器可以代替地址寄存器An来使用。

### 指令集

所有的68000指令在68020中都是可用的，而且其中的一些指令还特别灵活。例如，转移指

令可以是32位的偏移量,有几条指令可以选用更长的操作数。指令集中还提供了一些新指令。比如进行位字节操作的指令。

#### 片上高速缓存

68020芯片中包含一个很小的指令高速缓存(cache),这个Cache只有256个字节,可构成64个长字块,采用直接映射方式向Cache中装入新的字。

### 11.2.2 68030和68040处理器的改进

68030与68020的主要区别有两方面。68030中除了具有指令高速缓存以外,还有一块同样大小的高速缓存用来存放数据。数据高速缓存有16个块,每块4个字长。68030中还包含一个存储器管理单元(MMU)。

68030中的执行单元产生虚拟地址。高速缓存访问电路根据虚拟地址来判断所需要的操作数是否在高速缓存中。MMU将虚拟地址转换成物理地址并且和高速缓存的访问同时进行,这样一旦高速缓存访问失败,物理地址可以直接从内存中访问到操作数。

584 68040中包含一个浮点单元,该单元在第6章中描述过,它实现了IEEE的浮点标准。与68030相同,68040中也包含指令高速缓存和数据高速缓存。其存储器管理比68030更加完善;68040中有两个独立的地址转换高速缓存允许同时进行指令和数据的地址转换。还有一个流水线结构允许在前面的指令还在处理时进行取指令操作。两条与两个高速缓存相连的内部总线分别用来进行相应的指令和数据传输。这两条总线与两个地址转换电路相连,允许同时访问指令和数据高速缓存。

最后,68040中包含监视外部总线行为的电路。这个特点使得68040很适合用于多处理器系统。在多处理器系统中的一个关键要求就是保持不同处理器高速缓存中暂存的共享数据的一致性。关于如何通过总线监视电路来检测改变缓存数据的总线传输,我们将在第12章中描述。

### 11.2.3 68060处理器

最新的680X0成员是在90年代中期提出的68060<sup>[6]</sup>,它的时钟频率是50到75MHz。这款处理器主要是面向嵌入式系统市场,新的组织结构和制造特点使之在性能上是40MHz 68040的2.5倍。

68060是一个基于流水线的超标量处理器。流水线有四个基本段和两个附加段,其中的两个附加段在存储器需要回写时使用。该处理器每个时钟周期可以发出三条指令。三个功能单元——两个整型单元和一个浮点单元——构成主要的指令处理硬件。处理器中有单独集成在芯片上的8K字节的指令和数据高速缓存。每个高速缓存是4路组相联装置并使用16字节的块。两个64入口,4路组相联装置,便于从虚拟地址到物理地址转换的转换监视缓冲区,这些部件中都提供了高速缓存。该处理器采用动态的转移预判方法,使得通过流水线方式读取的指令流更加流畅。

### 11.2.4 ColdFire系列

从20世纪90年代中期开始, Motorola已经生产了一系列处理器部件和小型结构计算机,被称为ColdFire系列,该系列处理器主要是面向嵌入式系统市场的。它以68060处理器内核为基础。为了进行并行或串行连接,许多不同的产品中配置了一些存储器或I/O端口硬件。这些产品是为了满足各种应用的不同功能要求和性能要求。在ColdFire系列中,硬件芯片产品和综合软件设计都是可实现的。

## 11.3 Intel IA-32系列

Intel处理器<sup>[7]</sup>广泛应用于笔记本电脑和个人计算机中，并且事实证明已经取得了很大的商业成功。在20世纪80年代，Intel生产的首批处理器用在IBM PC中。它是在8086处理器的基础上于1979年诞生的，外部地址宽度是20位，操作的内外数据宽度是16位（有趣的是8086系列中称为8088的8位版本，实际上首次在IBM PC上是为了尽可能降低成本而采用的）。由于8086的封装具有40个引脚，所以地址和数据的传输在芯片的同一个引脚上通过多次来完成。

585

通过对相同基本指令集体系结构实现的发展和完善，相继问世了功能更强的处理器。它们是80286、80386和80486以及当前的Pentium系列。其中，80286是16位处理器。其他几种在处理内外部数据和地址时宽度都是32位的。80386是在IA-32系列中的第一款处理器。32位芯片带来了更大的封装，这样可以避免对于地址和数据线多路复用的需求。

### 11.3.1 IA-32存储器分段

在3.16.1节中，我们简要讨论了使用段寄存器（参见图3-37）来产生IA-32体系结构的存储器地址。这里将对此进行详细阐述。首先，要指出的是最初在8086中是怎样使用段寄存器的。由于当前的IA-32处理器可以转变成段寄存器操作的一个状态，称为实模式，因此，当前的IA-32处理器具有可以运行8086机器代码程序的功能。

#### 实模式

由8086处理器使用的这种地址生成模式，将存储器看成由段构成，每个段有64K字节。一个64K的存储器段按8086寻址方式可以用16位的有效地址在内部生成。处理器使用CS、SS、DS和ES段寄存器来分别访问代码段、堆栈段和两个数据段。在80386中还增加了另外两个段寄存器（FS和GS）。

586

图11-1给出了20位外部存储器地址的生成。16位的段寄存器值左移四位形成20位的地址，该地址作为段的起始地址。由8086寻址方式产生的16位有效地址在图中标有偏移量，它和段起始地址相加产生实际需要的20位存储器地址。

我们只要将地址中的高16位作为该段的起始地址保存到相应的段寄存器中，就可以将段放在20位地址空间中的不同位置上。20位的地址总共可以生成16个64K的非重叠段，可寻址的存储器总容量是1M。注意，段是可以重叠的。这样可以在不同程序之间进行指令和数据的共享。CS和SS的使用是由指令和所涉及的堆栈访问自动决定的。访问数据时默认使用的是DS段寄存器。在使用ES寄存器进行数据访问时指令中要加入前缀字节码。

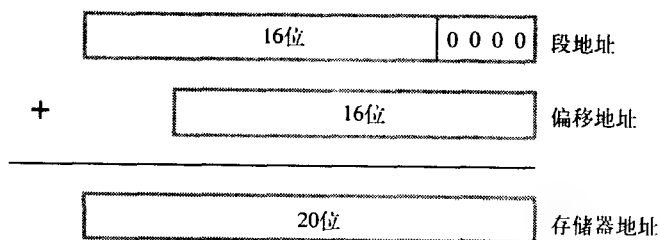


图11-1 8086处理器中存储器地址的生成

#### 保护模式

IA-32体系结构处理器通常采用保护模式来产生存储器地址。图11-2给出了最常用的利用基



址和变址寄存器中的内容,以及指令中的偏移量来产生物理地址的方式。一个32位的有效地址通过将变址寄存器的内容和比例因子1、2、4或8中的一个数相乘,所得结果再与基址寄存器内容和偏移量的和相加而得到。从六个段寄存器中选定一个段寄存器(如图3-37所示),用它的高14位来指定一个选择器,该选择器作为一个段描述符表的变址,并通过该变址得到一个32位的基地址。这个地址再加上分段单元中的有效地址产生一个32位的线性地址。最后,由分页单元利用页表将线性地址转换成相应的32位物理地址。

587 段描述符表和页表都很大,所以一般放在内存中。为了确保快速地完成地址转换,采用了转换监视缓冲区,这在第5章中介绍过。段描述符表包含访问权限域和指定一个段的最大空间的段限制域。这些参数由操作系统进行管理,这样可以在不同的应用程序同时占用内存时起到保护作用,因此将其命名为“保护模式”。

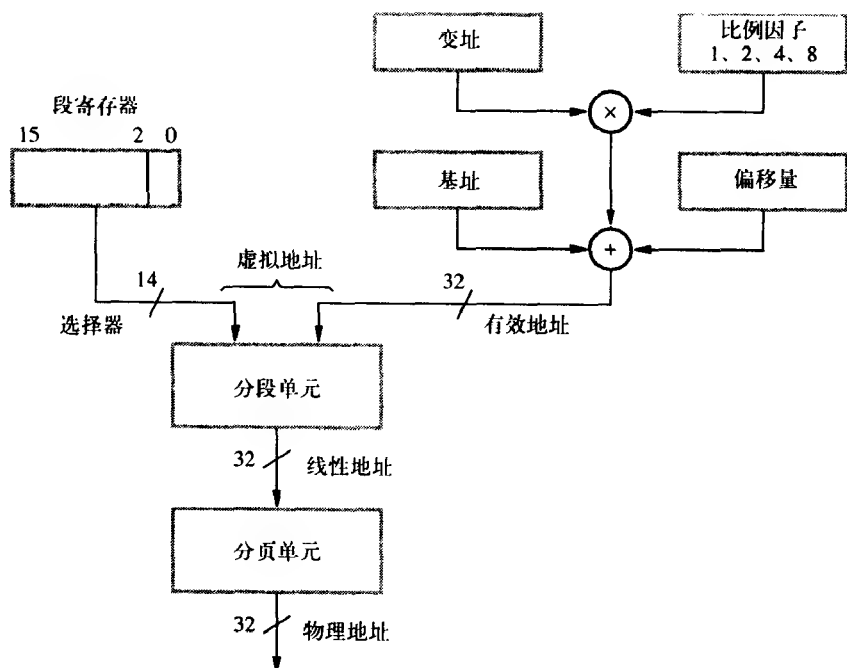


图11-2 IA-32体系结构中的地址生成

分段和分页的特点可以按以下方式中的任何一种方式构成存储器:

- 作为32位寻址空间中的一片使用,其有效地址当作物理地址使用。
- 用作一个或多个可变长度的段(没有分页)。
- 将32位的空间划分成一个或多个4K的页。
- 将分段和分页结合起来作为一种结构使用。

### 11.3.2 16位模式

IA-32处理器可以运行在一种特殊的模式下,该模式可以直接执行早期的16位Intel处理器(8086和80286)编制的机器语言程序。在这种模式下,仅仅使用处理器寄存器中低位部分,在图3-38的处理器中标志为AX, CX, …。处理器内部地址位宽是16位,使用的寻址方式是表3-3

中描述的子集。例如, 其中变址寄存器中的值不能用来生成有效地址。

32位和16位之间的转换操作实际上是通过逐条指令来完成的。还可以独立选择地址和数据的长度。例如, IA-32的所有寻址方式都可以和16位数据进行联合处理。程序在开始执行时采用的是默认模式。若要转换成其他模式, 只需在指令中加入一个前缀字节即可。这一点在图3-41中没有给出。

### 11.3.3 80386和80486 处理器

在第3章中介绍过80386是最先实现IA-32体系结构的处理器。它支持在第11.3.1节中介绍的存储器分段和分页技术。

80486处理器是最早与Motorola 68040具有相同数量(100万个)的晶体管之一。该处理器相对于80386在性能上的重要改进是进行电路的扩展。80486包括整型和浮点处理单元。整型单元是与80386处理器完全兼容的。浮点单元实现了在第6章中介绍的IEEE浮点标准。基于80386的计算机可以通过使用协处理器芯片而具备相同的功能。80486中的分页和存储器管理与80386不同。

588

在80486中包括一个4路组相联的指令和数据高速缓存。在向高速缓存中装入新信息时为提高效率, 采用批量数据传送机制, 该机制能一次将四个32位的字作为一块装入高速缓存中。高速缓存还有直接写的特点, 任何写入高速缓存中的数据同时也写到内存中。

为了具有更高的性能, 80486采用了并行机制和流水线处理方法, 并取得了明显的效果。整型单元和浮点单元可以并行执行指令。当执行一条指令时, 就从内存中取出后续的指令。频繁使用的指令执行时所用的时钟周期比在80386中要少。而对于装载和存储数据, 或执行寄存器到寄存器的操作仅仅需要一个时钟周期。

### 11.3.4 Pentium处理器

1993年问世的Pentium处理器<sup>[8]</sup>的性能相对于80486而言有了极大的提高。80486具有100万个晶体管, 而Pentium中达到了300万个。如果以程序中的整型运算为基准, 它的计算能力是80486的两倍以上; 如果以程序中的浮点运算为基准, 它的计算能力大约是80486的五倍左右。

Pentium处理器采用的是CISC体系结构, 并采用了很多RISC处理器结构的特点, 使其性能得到了很大的提高。这些性能在80486中还只是低程度的实现, 特别是用于指令和数据的独立专用的8K片上高速缓存。该超标量处理器由多流水线操作单元支持, 每个时钟周期可以发出两条指令。由于外部数据总线是64位宽, 所以可以很快地将主存中的数据装入高速缓存中。高速缓存是具有32字节块的2路组相联装置, 其包括三个独立流水线的操作单元——两个用于整型运算, 一个用于浮点运算。每个整型单元流水线深度是5段, 浮点单元流水线深度是8段。

Pentium处理器采用简化形式的动态转移预判技术, 选择最后一次执行的转移方向作为程序的执行方向。为此, 该处理器中采用了地址表, 用来保存每次转移指令最后转移的地址值。如果从转移开始到循环退出, 程序循环中所有转移都是正确的, 那么预判的方向就是正确的。

### 11.3.5 Pentium Pro 处理器

1995年问世的时钟频率是133MHz的Pentium Pro<sup>[9, 10]</sup>在性能上较时钟频率为100MHz的Pentium提高了近两倍。该处理器性能的提高主要取决于增加了超标量因素以及具有乱序执行指令的能力。超标量因素, 或者是在一个时钟周期内能完成的指令的最大数量, 在Pentium处理器

中是每个时钟周期两条指令，而在Pentium Pro中达到了三条。在多执行单元中的流水线深度达到了12，而在Pentium中只达到5；处理器内部数据通道的宽度是64位，是Pentium的两倍。在Pentium Pro中具有容量和Pentium相同、每片8K的独立片上一级（L1）指令和数据高速缓存。它还有一个容量是256K的二级（L2）高速缓存。这个高速缓存在处理器芯片的同一个包中，但是它在一块单独的芯片上通过一条64位总线和处理器芯片相连。

[589]

处理器中的多指令单元支持超标量操作，包括两个整型运算和两个浮点运算。Pentium Pro性能提高的主要原因是它可以用不同的顺序执行指令，而不按照存储在内存程序中指定的顺序来执行。这一特点使得它可以将更多的指令进行并行处理。当然这样做必须要提供相应的控制来保证程序执行的计算结果是正确的。在Pentium Pro中也实现了动态转移预判功能，这与Pentium相同。这种能力使得处理器可以在指令流中具有更大的处理范围，从而可以充分利用并行处理。

外部总线检测线路使得Pentium Pro适用于多处理器系统。这些检测线路是用来设置和保持存储在不同处理器高速缓存中共享数据的一致性的，并在适当的时候采取相应的控制动作（参见第12章“高速缓存一致性”部分的描述）。

### 11.3.6 Pentium II 和Pentium III 处理器

Pentium II处理器增加了在3.23.2节中描述的MMX指令。这种指令可以很方便地并行处理多媒体中用像素来表示图形的这种较短数据。通常，这些数据采用8个相同的保存浮点数的64位寄存器来进行存储。Pentium II中L1级高速缓存的容量是Pentium Pro中L1级高速缓存容量的两倍，每个高速缓存的容量是16K。片外的L2级高速缓存的容量是512K。

Pentium III 处理器引入了在3.23.3节中介绍的向量（SIMD）指令。这些指令，称为数据流SIMD扩展（SSE）指令<sup>[1]</sup>，用于高效处理操作数是浮点数的向量操作。四个32位的浮点操作数被分别打包装入到八个新的128位寄存器中，这些寄存器称为XMM寄存器。高速缓存中除带有一个额外的选项之外，其他都与Pentium II相同。Pentium III处理器的一个版本在相同的芯片上有256K的L2级高速缓存，它可以为L1级高速缓存提供更高带宽的通道。

在Pentium、Pentium Pro、Pentium II、Pentium III处理器分别于1993年、1995年、1997年和1999年问世时，它们的时钟频率分别是60、200、266和500MHz。随着电路技术和VLSI制造技术的提高，并进一步减小了晶体管的体积和门电路的延迟，在Pentium III处理器目前版本中其时钟频率可以达到1GHz。

### 11.3.7 Pentium 4 处理器

2000年问世的Pentium 4最初版本的时钟频率为1.3~1.5 GHz<sup>[7]</sup>。它支持全部IA-32指令集以及MMX和SSE指令。SSE指令已经扩展成为SSE2，能处理两包64位浮点数或128位寄存器中的两包64位整型。在实现数据安全应用中的加密和解密操作时，长整型发挥了很大的作用。Pentium 4中的流水线更深——相对于Pentium III 中的10段而言已经达到了20段——为了达到更高的时钟频率，它采用了更短的段，同时还改善了电路和制造技术。

[590]

Pentium 4处理器有两个独立的L1级的数据和指令高速缓存。数据高速缓存的容量是8K，用4路组相联装置可以访问64字节高速缓存块。指令高速缓存组织成可以保存已经解码的指令执行路径段，称为跟踪，它可以扩展原程序中的多条转移。如果路径是重复的，那么执行速度更快。当然，当跟踪重复时必须检查所选择的转移是否相同。术语跟踪高速缓存是用来描述这种跟踪

策略的。解码后的指令通常表示成微操作。凑足四条微操作就将它们表示成一条IA-32指令。跟踪高速缓存可以保存很多执行路径段,这些段总共由12K微操作构成。

256K字节的片上L2高速缓存设计成128字节块和8路组相联装置。L2和L1级高速缓存之间的传输路径支持的传输速率是48GB/s,在Pentium III中只有16GB/s。

Pentium 4中的系统总线比Pentium III中总线提供了更高的传输率。Pentium 4的系统总线是64位宽,可以在400MHz时进行传输,因此传输率是3.2GB/s,而在Pentium III中只有1GB/s。

### 11.3.8 AMD公司的IA-32处理器

除Intel公司外,其他公司也生产实现了IA-32体系结构的处理器,目的是为了在个人计算机和工作站上与Intel公司的相应产品进行竞争。AMD公司<sup>[12]</sup>曾经一段时间就曾如此。在2000年,AMD的Athlon处理器就可以达到时钟频率为1.2GHz,在性能上是与Pentium4相当的产品。

Athlon是在处理器芯片上采用两级的缓存技术。L1级的高速缓存容量是128K, L2级高速缓存的容量是256K。一个双倍速率的DRAM内存(参见第5章)的接口可以提供最高的传输率是2.1GB/s。系统I/O总线协议可以在200MHz和266MHz下工作。

## 11.4 PowerPC 系列

在20世纪90年代早期,IBM、Motorola和苹果公司为了面向个人计算机和工作站市场共同合作开发了RISC型处理器系列PowerPC<sup>[13, 14]</sup>。由IBM和Motorola<sup>[15]</sup>合作生产的PowerPC处理器被应用在IBM和苹果计算机上。通常,这些处理器具备的体系结构特征与Intel IA-32处理器所提供的计算能力相当。第一台实现PowerPC体系结构的处理器是1993年生产的601。首先我们总体看一下该系列指令集的体系结构,之后再描述其中一些处理器的实现。

### 11.4.1 寄存器集

该处理器中有32个通用寄存器和32个浮点寄存器。浮点寄存器的宽度是64位,采用IEEE标准表示浮点数。PowerPC体系结构定义了32位和64位的操作模式。通用寄存器的大小由特定处理器的实现模式来决定。

591

### 11.4.2 存储器寻址方式

存储器是按照字节寻址的,并且只使用Load和Store两条指令在存储器和处理器寄存器之间进行数据的传输。为了保持RISC的设计类型,只采用了简单的变址寻址方式。有效地址通过在基址寄存器中的内容加上变址产生,变址可以是指令中的立即数或是在基址寄存器中的内容。作为可选操作,有效地址可以写入基址寄存器中。这就使得在连续的内存地址中可以很方便地读写操作数序列。在传送乘法操作数时可以使用单指令。Load 和Store指令有很多版本,在传送不同类型和大小的操作数时提供了很大的灵活性。

### 11.4.3 指令

PowerPC指令长度是32位的固定格式。算术和逻辑指令使用3寄存器格式,两个操作数寄存器和一个用来保存结果的目标寄存器。还提供了大量的条件转移指令。PowerPC中提供了“乘加”

指令, 对保存在寄存器RA、RB和RC中的浮点数进行以下操作:

$$RD \leftarrow ([RA] \times [RB]) \pm [RC]$$

其中的一类条件转移指令对计数器减1然后再转移, 但减1操作是否执行要看计数器中的值是否减到0。乘加操作、减1转移、多个操作数的传送指令和在变址寻址中可选择的修改基址寄存器操作都是典型RISC ISA中的特点。这四个特点分别在终止循环、进入和退出程序时保存和恢复处理器寄存器以及处理表中的数据项中都是非常有效的。PowerPC的设计者在综合这些特点时采用了适当的折衷, 保留了高效率、简单的数据流和流水线指令, 这都是RISC机器中的基本特点。应该注意的是所有的这些特征中, 除了减1转移操作外, 其他的在ARM ISA中都存在相同的形式。

#### 11.4.4 PPowerPC处理器

在IBM的系列产品中, PowerPC体系结构的计算机继承了使用IBM 精简指令集系统 (RS) /6000系列处理器的POWER体系结构。率先实现PowerPC体系结构的601处理器是两种体系结构之间的过渡产品; 同样, 它实现了POWER和PowerPC指令集的全部内容。这样, 601就可以同时编译POWER程序和PowerPC程序。在系列中的后续处理器就是纯粹的PowerPC处理器了。

592

##### PowerPC 601处理器

601处理器芯片包含280万个晶体管, 它最先应用于IBM台式机中。601是一个32位处理器, 主要面向笔记本、台式计算机和低端多处理器系统。相应不同版本的处理器的时钟频率分别有50MHz、66MHz、80MHz和100MHz。

PowerPC 601的处理器芯片上有一个用来存储指令和数据的32K高速缓存。高速缓存设计成8路的组相联装置。它提供三个独立的执行单元: 一个整型单元、一个浮点单元和一个转移处理单元。为了达到超标量运算能力, 一个时钟周期内可以发出三条指令。601中的整型指令有四个流水线阶段, 浮点指令有六个流水线阶段。

##### PowerPC 603处理器

603处理器的处理宽度也是32位。它主要面向笔记本和台式计算机, 是低成本、低功耗处理器, 在80MHz时的功率是3瓦特。它提供的五个独立执行单元可以并行处理, 每个时钟周期发出三条指令, 所以指令发布和控制硬件相应要比601复杂。芯片上的高速缓存分成两个8K字节的单独部分, 分别用来存储指令和数据。

##### PowerPC 604处理器

604处理器也是32位的, 它的性能要比601和603都高, 其中整型和浮点的运算速度是601和603的两倍。604处理器的性能已经达到时钟频率100MHz、每个时钟周期内能发出四条指令的超标量能力。处理器中有六个独立处理单元: 三个整型单元、一个浮点单元、一个存储器读/写单元和一个转移处理单元。这种处理器主要面向个人计算机和中等规模的工作站。

##### PowerPC 620处理器

620处理器实现了完全64位PowerPC体系结构并具有超标量的性能。它主要面向高档的台式计算机、服务器、事务处理系统和多处理器系统。

593

与604相同, 620有六个单独的执行单元, 一个时钟周期内能执行四条指令。由于处理器可以乱序执行指令, 特定程序中指令执行的实际效率会大大提高。该处理器采用了动态转移预判

断的技术, 处理器芯片包含指令高速缓存和数据高速缓存。每个高速缓存容量是32K, 并采用8路组相联装置。

### MPC7450 处理器

601、603、604和620是由IBM生产的第一批PowerPC系列产品。在6XX系列之后, Motorola实现了7XX和7XXX的PowerPC处理器, 并冠以前缀MPC。在MPC7XXX系列中最新的处理器是2001年年初问世的MPC7450, 它的时钟频率达到733MHz。该款处理器用在Apple Power Mac G4系列计算机中。

MPC7450是7段流水线的超标量处理器。一个时钟周期内能将四条指令发送到功能单元中。这样的功能单元共有十一个: 一个读/写单元, 一个转移单元, 四个整型单元, 一个浮点单元, 还有四个单元用来执行打包向量数据操作数的并行算术运算。Motorola对这些后面的单元都用AltiVec<sup>[16]</sup>命名。

AltiVec硬件对向量操作数进行并行处理, 这与Intel Pentium处理器进行MMX和SSE操作是相同的, 就像我们在3.23.2和11.3.6节中描述的那样。AltiVec指令所操作的打包数据存储在32个128位的向量寄存器中, 这些寄存器与通用寄存器和浮点寄存器是独立的。一个向量寄存器可以存储16个8位整数、8个16位整数、4个32位整数或4个单精度(32位)浮点数。向量的Load和Store指令用来在内存和向量寄存器之间传送数据。AltiVec指令加速了多媒体和信号处理操作。指令中有一条是乘法累积指令, 用来将两个向量寄存器中的对应元素相乘之后, 再将结果和第三个向量寄存器中对应的元素相加。这种操作在数字信号处理中是非常普遍的。另外, 该处理器中还包含向量点乘积指令。

片上的L1级高速缓存组成了独立的32K指令和数据高速缓存。这些高速缓存采用8路组相联装置。L2级高速缓存也在片上, 其容量是256K, 采用8路组相联装置。在处理器的时钟频率下, L1和L2级高速缓存之间的数据传送宽度是256位。片外的L3级高速缓存通过64位总线进行访问, 它可以配置的容量是1M或2M。

## 11.5 Sun公司SPARC系列

由Sun公司开发的SPARC体系结构是一个可升级的体系结构。它由一系列性能逐渐提高的处理器来实现技术和组织上的创新发展。基本指令集的体系结构保持不变。SPARC处理器针对高性能的工作站和服务器的, 主要用于处理器内存共享的多处理器系统中。我们将在第12章介绍这类系统。

SPARC体系结构是使用3寄存器、32位固定长度指令格式的RISC型体系结构。在指令中指定两个源操作数寄存器和一个目标寄存器。所有对数据进行运算的指令操作都是在处理器的寄存器中完成的。处理器中有32个通用整型和地址寄存器, 32个浮点操作数寄存器。可以访问内存的指令是Load和Store指令, 主要用来在内存和寄存器之间进行数据的传送。

SPARC体系结构于1987年首次实现, 它有32位地址和32位数据。该体系结构的最新版本是版本9, 它处理的地址和数据宽度都是64位, 由UltraSPARC系列处理器实现。指令宽度还是32位, 所有寄存器的编程模式保持不变。它保持向后兼容性, 也就是说, UltraSPARC处理器能执行早期版本体系结构的机器代码。

我们在第8章中引入SPARC体系结构。这里用UltraSPARC处理器来说明在高性能处理器中怎样实现流水线功能。UltraSPARC系列中包括UltraSPARC I、II、III, 其特点是多处理单元和超标

量性能。除了基本的SPARC指令集以外,又引入了很多特殊的指令来支持图形和多媒体的应用。这些称为可视化指令集(VIS)。VIS指令提供了对图形像素或数字信号样本打包成64位字的异行向量操作数。VIS指令与Intel的MMX和SSE指令(第3章部分III和第11.3.6节)及Motorola AltiVec(第11.4.4节)指令很相似。UltraSPARC系列中的后续成员逐渐提高了性能,即具有更高的时钟频率、更快的内存和I/O接口以及更大和更加复杂的高速缓存结构。例如,UltraSPARC I采用0.5微米的CMOS技术,时钟频率是167MHz<sup>[17]</sup>。它的后续产品UltraSPARC II,同样包括9段流水线结构,但由于采用了更快的0.25微米的技术<sup>[18]</sup>而使性能进一步提高,其运行的时钟频率在250MHz到480MHz之间。

最新的系列成员UltraSPARC III,采用14段的流水线结构<sup>[19]</sup>。处理器中有四个整型执行单元和三个浮点单元,还可以处理VIS指令。UltraSPARC III采用0.18微米的制造技术,时钟频率在750MHz到900MHz之间,今后可以制造时钟频率为1.5GHz的处理器。片上的L1数据高速缓存的容量是64K,指令高速缓存的容量是32K。它们都是4路组相联装置,并对32字节的块进行操作。外部的L2级高速缓存采用直接映射的方法,可以配置的容量是4M或8M。UltraSPARC III还支持多处理器配置,同时具有连接几百个处理器的潜力。

#### microSPARC 系列

**[595]** 另一个基于SPARC体系结构的处理器系列称为microSPARC。这个系列中的成员是基于SPARC体系结构说明书中的版本8开发的32位处理器。该处理器主要针对低成本单处理器的应用。该系列成员中的某些处理器,如microSPARCIIep,还包括一个处理器片上的PCI接口和存储器控制器,很适用于嵌入式应用。这些微处理器与UltraSPARC处理器的完全兼容性为开发嵌入式的应用提供了很大的便利。这样一个特定的应用软件可以在功能强大的工作站上开发和测试,然后在开发的最后阶段将软件下载到目标处理器中。

### 11.6 康柏ALPHA系列

DEC公司在1992年推出的Alpha体系结构是32位VAX系列<sup>[20]</sup>的后续产品。康柏公司在1998年收购了DEC公司。该公司采用Alpha处理器生产了一系列高档工作站和服务系统,并用序号21X64来标注,X=0、1和2。

Alpha体系结构采用地址和数据宽度都是64位的RISC设计。它有32个通用寄存器和32个浮点寄存器。在所有的21X64处理器中都采用了多流水线处理单元,以达到超标量指令的执行效率和提高静态和动态转移预判断的处理能力。它使用了片上的独立数据和指令高速缓存。

流水线处理器设计的基本目标是在给定任何一种实现技术和较高的时钟频率条件下,尽可能保持每个流水线有较短的逻辑深度,以此来减小阶段间的延迟。Alpha体系结构最重要的设计特点是使用简单的指令格式和寻址方式,以达到缩短流水线段之间的延迟。为了使L1级高速缓存和处理器之间的数据传送延迟达到最小,只采用32位和64位的Load和Store两条指令。

#### 11.6.1 指令和寻址方式的格式

Alpha ISA只有四种指令类型,长度全都是32位:

操作——在这种类型中包括整数、浮点数和字节操作的运算。这些指令采用三操作数格式,两个操作数在处理器的寄存器或指令的立即字段中。

内存——Load/Store操作采用寄存器加变址地址的偏移量作为惟一的寻址方式。

**转移**——条件转移指令包含一个偏移量值，该值用来指定方向和与程序计数器相关的转移目标地址的距离。这里没有条件码寄存器，条件代码通过操作指令可选地写到通用寄存器中。该寄存器由需要测试代码的转移指令进行命名。无条件转移指令用命名的寄存器来保存修改的程序计数器的值，以便在转移程序是子程序调用时将该值作为返回地址。

596

**调用PAL**——特权体系结构库 (Privileged Architecture Library, PAL) 指令执行在用户模式下不可见的操作系统功能。这些特权指令可以访问那些普通指令集不能访问的硬件资源，即处理器状态寄存器。PAL程序还包含一些在Alpha指令集中不存在的指令。它们主要用于中断和对存储器管理单元的寄存器进行操作。

### 11.6.2 ALPHA 21064处理器

Alpha体系结构最先是由21064实现的，它是有170万个晶体管、功率为30瓦、时钟频率为200MHz的芯片<sup>[22]</sup>，有8K字节的指令和数据L1级高速缓存。两个高速缓存都是直接映射的32字节块。外部的L2级高速缓存可以配置的容量是128K到8M字节。存储器管理单元有单独的转换监视缓冲区来监控指令（12入口）和数据（32入口）的访问。

每个时钟周期内最多发出两条指令，使用四个独立的处理单元：一个整型单元、一个浮点型单元、一个转移单元以及一个存储器读/写单元。四个单元的流水线深度分别是7、10、6和7。前四个段是公用的，可以并行处理两条指令。

### 11.6.3 ALPHA 21164处理器

21164处理器在1994年推出<sup>[23]</sup>。它的性能是最初21064处理器的两倍。21164中的晶体管数量达到了930万个，在时钟频率是300MHz时的功率为50瓦。提供片上统一的96K的L2级高速缓存以及8K的L1级指令和数据高速缓存。L2级高速缓存是3路组相联装置，并有64字节的块。片外的L3级高速缓存可配置的容量是1M到64M字节。

每个时钟周期内最多发出四条指令，是21064的两倍。21164比21064多了一个用来管理L2和L3级高速缓存的功能单元。两者的流水线深度相同。存储器管理硬件设置了一个48入口的转换监视缓冲区来访问指令，以及一个64入口的缓冲区来访问数据。

### 11.6.4 ALPHA 21264处理器

21264是21X64系列<sup>[24]</sup>中最新的处理器。它是在1998年推出的，时钟频率为500MHz。在2001年年初，运行的版本已经达到了850MHz。处理器芯片中有1500万个晶体管。

与早期的Alpha处理器相比，高速缓存方案有了非常大的改变。L1指令和数据高速缓存比以前大很多，每个都是64K，都是2路的组相联装置。片外统一的L2级高速缓存可以配置的容量是1M到16M。21264没有L3级高速缓存。由于有了更大的L1级高速缓存，从而增加了命中率，这样即使L2级高速缓存是片外的，也会使存储器的访问延迟整体减少。

597

21264和早期Alpha处理器的另一个重要的不同之处在于，它可以将指令乱序发送到功能单元中，就像第8章中讨论的那样。这对于典型程序可以增强它超标量指令的执行效率。每个时钟周期内发出四条指令，这与21164中的相同。但是由于对整型和浮点单元中部分内容的复制导致了功能单元资源的增加，同时还增加了一个新的功能单元来处理视频数据。这一指令执行硬件的提高，再配合更高的时钟频率和乱序发布指令的能力，使得它的性能比21164提高了两倍。



## 11.7 Intel IA-64系列

从20世纪90年代中期开始, Intel和Hewlett-Packard联合开发了64位被称为IA-64<sup>[25]</sup>的微处理器体系结构。最早实现该体系结构的处理器是Itanium (更早的代号是Merced)。IA-64体系结构与IA-32体系结构完全不同, 它采用Intel的体系结构, 支持以前的80386以及持续发展的Pentium系列处理器。Intel希望一直都生产IA-32和IA-64体系结构兼容的处理器。IA-64体系结构的编程特点在参考文献[26]和[27]中有详细描述。

IA-64体系结构有64位的地址空间和64位的整型和浮点格式, 类似于RISC的3寄存器指令格式占41位。该结构为128个通用寄存器或128个浮点寄存器寻址提供了三个7位的寄存器域。下面将介绍一个6位域指定指令的条件执行。指令中剩下的14位用来指定OP码。

### 11.7.1 指令包

IA-64体系结构中的一个显著特征是, 三个41位的指令再加上5位称作为模板的域组成了一个128位的包(bundle)。模板用来说明怎样由编译器指定对指令进行并行处理的信息。例如, 有一个模板代码指定了一个停止位, 这就表示在该组指令结束时可以进行并行处理。这样的组可以扩展很多的指令包。模板中的信息是处理器用来对多功能单元中的指令进行调度以便并行处理, 从而实现超标量的操作。IA-64的这个特点称为式并行指令计算(EPIC)。EPIC可以认为是对超长指令字(VLIW)指令集设计<sup>[28]</sup>的一种扩展。VLIW体系结构中, 每条指令指定一些可能的不同操作, 这些操作可以用在并行独立的数据操作中。

### 11.7.2 条件执行

IA-64体系结构中的一个主要特征是在指令中使用条件执行, 称为预判。每条指令中64位的预判域是从处理器64个1位预判标志中选择的。这些标志有效地代替了传统处理器中的条件代码标志。如果名为flag的值是1则指令执行; 否则, 该指令不执行。实际上, 指令是通过指令流水线来执行的, 如果预判标志是1时才将结果写到目标寄存器中。这个特点很像在ARM体系结构指令的条件执行, 我们在第3章的部分I中描述过。

指令的条件执行通过将条件转移转移到某个特定的位置上, 以达到提高程序指令执行的效率。例如, 一条简短的前向条件转移通过执行它的条件代码, 删除该条件转移不成立时的位置到其目标位置之间的代码。代码块中保护每条指令执行的预判标志通过提前对该代码块进行测试和比较来设置。

在生成IA-64代码中的一个if-then-else结构时也可以进行性能改善。图11-3a给出了在传统的机器代码中, 如果寄存器R1和R2的内容相等就执行加法指令, 如果不相等则执行减法指令。在图11-3b中给出了相应的IA-64代码。IA-64比较相等指令的操作过程如下: 如果R1和R2的内容相等, 预判标志P1就是

	Compare	R1,R2
	Branch $\neq 0$	ELSE
THEN:	Add	R3,R4,R5
	Branch	NEXT
ELSE:	Subtract	R6,R7,R8
NEXT:	...	

a) 传统的代码

	CompareEqual	P2,P1 = R1,R2 ;;
	(P1) Add	R3,R4,R5
	(P2) Subtract	R6,R7,R8 ;;
NEXT:	...	

b) IA-64 代码

图11-3 在IA-64体系结构中实现if-then-else代码

1; 否则, 预判标志P1是0。标志P2是对P1的补充。如果P1=1, 执行加法指令; 如果P2=1, 执行减法指令。双分号表示停止位置。在此例中, 两个双分号之间的加法和减法指令可以进行并行调度处理。由P1和P2的值来决定哪个操作的结果将写到指定的目标寄存器中。在加法指令和减法指令到达写阶段之前, P1和P2的值还未确定, 那么指令执行流水线不会装入新的指令。

除了采用这类提高性能的技术以外, IA-64还采用了第8章讨论过的转移预判和推测性执行。

### 11.7.3 推测性装入

为了减少由寄存器访问内存装入指令而带来的延迟, 引入了一种称为推测性装入 (speculative load) 的特殊形式的装入指令, 它可以由编译器自动生成。该装入通常放在被编译程序的前面位置。这样做增加了当需要该指令时它在寄存器中的机会, 从而避免了相应访问内存的时间延迟。但如果我们使用该指令, 必须首先检查它确实在寄存器中。在处理推测性装入的指令是需要提前传送的预判转移指令时, 要给予特殊的考虑。

### 11.7.4 寄存器和寄存器堆栈

IA-64体系结构中指定了128个通用寄存器来存储64位的整数或地址, 此外还有128个用来保存双精度 (64位) 浮点数的寄存器。两个单精度数可以打包存在一个寄存器中。还有8个用来保存子程序call/return的连接地址的64位寄存器。

通用寄存器中的前32个R0到R31供一般的数据和地址使用。剩下的96个寄存器R32到R127作为寄存器堆栈使用, 用来保存子程序的局部变量, 以及在调用程序和子程序之间传递参数。这个寄存器堆栈有效地代替了处理器的堆栈, 并且是在内存中实现的, 这在2.9.1节中描述过。该寄存器堆栈的管理方式是, 寄存器不必向内存或从内存中保存/恢复一系列嵌套子程序的调用, 它具体的工作流程描述如下: 首先假定所有子程序所需要的全部和局部参数变量的寄存器空间不超过96。在发生子程序调用时, 处理器控制硬件自动从寄存器堆栈中“拿出”一部分到内存中, 创建所需的额外寄存器空间。当子程序返回时, 会自动将这部分空间归还给寄存器堆栈。

处理器还自动进行寄存器的重命名, 这样所有的程序——主程序和子程序尽管所用的物理寄存器不同, 但总是能指向它们自己局部的R32以上号码的寄存器 (寄存器重命名的另一版本在第8章中讨论过)。调用程序的局部寄存器的高地址部分和被调程序局部寄存器的低地址部分是重叠区域, 该区域用来实现参数的传递。

图11-4给出了当主程序调用一个子程序时如何管理寄存器堆栈。图中给出的堆栈寄存器的序号是向上递增的。这样容易与第2章所举的堆栈 (堆栈增加的方向是内存地址减小的方向) 例子进行比较。寄存器R0到R31是所有程序都可见的, 可以用来保存全局变量, 但是在图中没有给出。假定主程序使用从R32到R39的八个寄存器存储它的局部变量, 用R40到R43的四个寄存器来为子程序传递参数。通过在主程序中的如下指令来向处理器硬件声明寄存器的使用:

Alloc 8, 4

图11-4a中给出了它的状态。在主程序调用子程序之后, 子程序执行指令:

Alloc 7, 3

用来声明它需要七个局部寄存器, 四个用来接收从调用程序传送来的参数, 三个用来为第二个子程序传递参数。处理器硬件对十个 (物理的) 寄存器R40到R49进行重新映射, 使得它们可以

和子程序的R32到R41寄存器相对应。在执行子程序时活动的寄存器堆栈部分如图11-4b所示。执行时动态映射所需的信息是在主程序中执行Alloc指令时的值给出的。当子程序执行返回主程序时，寄存器堆栈的活动部分就恢复到图11-4a中所示的状态。寄存器堆栈实现了一个多寄存器窗口的版本，这一点在伯克利RISC设计<sup>[29, 30]</sup>和UltraSPARC II处理器中所采用（参见第8章）。

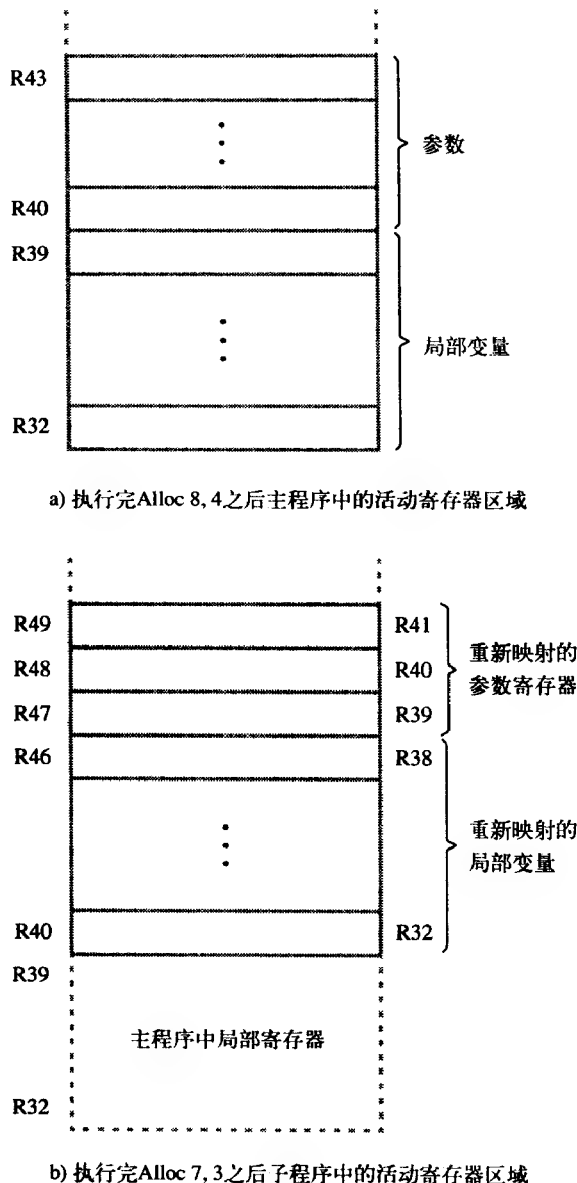


图11-4 IA-64中为局部变量和参数传递分配的寄存器堆栈

另一种寄存器重命名的形式称为寄存器循环，在IA-64中采用。它假定在重复的数据之间没有数据依赖性的限制，用来重复执行循环的逐次迭代。通常，一个Loop循环中连续重复使用相同的寄存器，这些寄存器是在循环体中进行命名的。在IA-64体系结构中，编译器为循环体生成一个副本，该副本允许硬件自动重命名寄存器，以便在循环连续的重复部分中使用不同的物理寄存器。这样就允许指令调度和执行硬件在前面的重复还没结束时就开始这次连续的重复部分

操作。这项技术降低了整个循环的执行时间，被称为软件流水线。它不同于循环展开，在循环展开中复制的循环体副本是使用机器指令代码来替代的。

### 11.7.5 Itanium 处理器

Itanium是最先实现IA-64体系结构<sup>[31]</sup>的处理器。它提供了更多的复制功能单元给不同类型的操作：整型、浮点、多媒体（类似于IA-32体系结构中的MMX操作，在第3章部分III中描述过）。在时钟周期是800MHz时可以向10段流水线发出六条（两个3指令包）超标量的指令。它的功能单元有：4个整型单元、4个浮点单元、4个多媒体（MMX）单元、2个读/写单元和3个转移单元。整型寄存器为能与多功能单元同时交换数据，在边沿上设有8个读口和6个写口。

602

处理器中有三级高速缓存单元。L1和L2高速缓存和处理器在同一块芯片上，L3在单独芯片上，不过这个芯片和处理器封装在同一个包装中。L1级高速缓存由单独的每个容量是16K字节的指令和数据高速缓存构成。它们是4路组相联装置，有32字节的块。指令高速缓存能在每个时钟周期内向处理器传送两条3指令包（256位）。L2高速缓存容量是96K字节，它是6路组相联装置，有64字节块。L3级高速缓存的总容量是4M字节，带64字节块。它是4路组相联装置，通过一条128位的内部总线与L2级高速缓存相连，在处理器时钟频率下的传输速率是12.8GB/s。

在高速缓存之间以及L1级高速缓存和处理器之间的交互，是在将平均指令执行速率下高速缓存失败和处理器流水线延迟减到最小的条件下进行的。这些交互中的一些特征是值得关注的。在L1级指令高速缓存和处理器之间有个解耦缓冲区，它可以容纳八个3指令包。这就允许在处理器发送指令的延迟期间可以从L1级高速缓存向缓冲区连续预取指令。相反，当高速缓存在预取的过程中失败时，处理器可以继续从缓冲区读取或发出指令。在L1和L2级高速缓存之间也有一个缓冲区用于L2从L1预取指令。这个缓冲区的大小是L1级高速缓存和处理器指令发布硬件之间缓冲区的两倍。L1级高速缓存只处理整型寄存器，浮点操作数直接从L2级高速缓存装入浮点寄存器中。

处理器中有一个64位的系统总线用来连接包，该包中包含处理器和高速缓存到其他系统部件如主存和I/O设备的连接。它支持266MHz的传输速率，也就是2.1GB/s。

外部总线控制器提供将四个Itanium处理器直接连接到一个多处理器的结构。当多处理器共享外部存储器单元时，控制器处理所需高速缓存一致性的操作。这些将在第12章中介绍。

### 11.8 堆栈处理器

本书中所讨论的所有处理器都是采用通用寄存器保存操作数的。指令可以按照任意的顺序访问它们。几年前，Hewlett-Packard（惠普）公司设计制造了一台名为HP3000的计算机，它主要的体系结构特征是有一套特别的指令，该指令处理的操作数是保存在堆栈数据结构中的。访问堆栈中的操作数要求只能对栈顶的数据进行操作，操作结果总是保存到栈顶。这种类型的设计不适合于当前高度并行的RISC和CISC处理器。在这些处理器中，当许多寄存器中同时访问几个操作数时要求有很高的性能。然而，HP3000和更早由Burroughs公司成产的B5500、B6500和B6700系列计算机也是面向堆栈处理的，作为堆栈式计算机的商品其具有重要的历史意义。这些机器处理算术表达式的方式既有趣又巧妙。我们通过描述HP3000指令集和寻址方式来说明它的主要思想。讨论主要集中在这种计算机堆栈组织结构的特点上。

603

## 11.8.1 堆栈结构

HP3000是16位计算机。程序中的指令和数据在内存中是分开存放的；而对于使用立即数的程序来说，指令和数据是不可混淆的概念。硬件寄存器用作程序和数据段的指针，如图11-5所示。

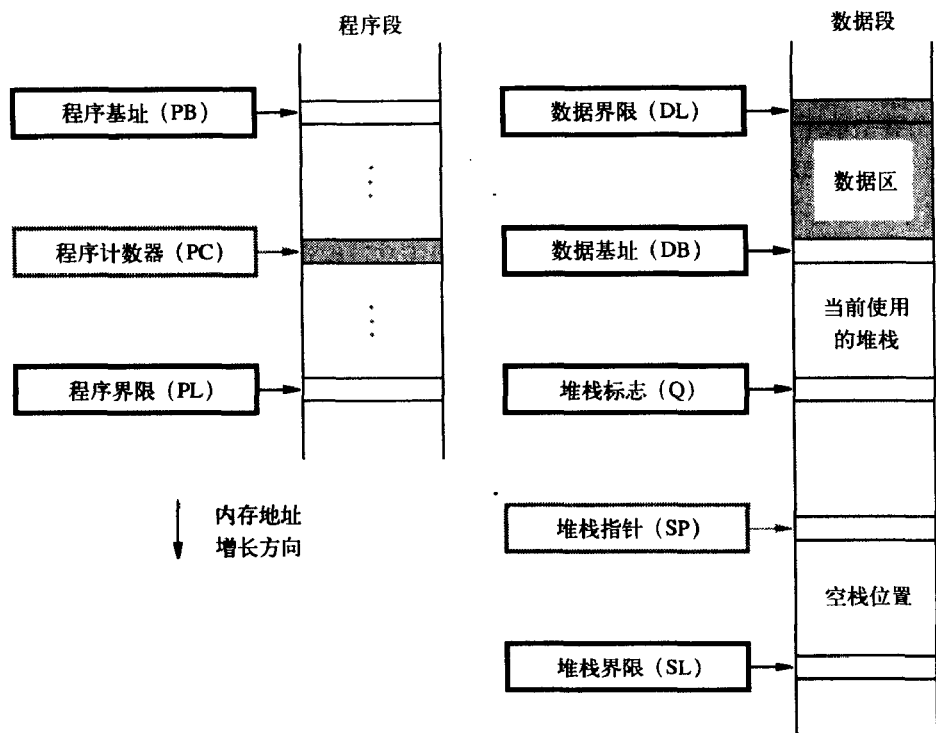


图11-5 HP3000中的程序和数据段结构

程序中的段用三个寄存器来指定。程序基址 (PB) 和程序界限 (PL) 寄存器表示程序所占内存空间的大小，程序计数器 (PC) 指向当前正在执行的指令。每一个寄存器都使用16位的地址。

数据段被分成两个部分——堆栈和数据区。五个16位指针用来表示和访问这些内存单元。数据基址 (DB) 寄存器的内容表示堆栈区域的起始地址。堆栈向高地址的方向增长。如果堆栈的栈顶元素在位置  $i$ ，那么下一个进栈元素的位置将是  $i + 1$ 。这一点和本书中讨论的其他堆栈方式正好相反，以前的堆栈是假定堆栈增长方向是向内存地址减小的方向。堆栈顶端元素的地址又称栈顶 (TOS)，存储在16位的堆栈指针 (SP) 中。SP实际上并不是像我们简单说明的那样是一个单一的硬件寄存器，但是也可以这样考虑。它会随着数据元素的压栈或弹出而增减。从用户的角度来看，它的功能和其他的16位指针寄存器相同。堆栈的上限由堆栈界限 (SL) 寄存器的内容指定。因此，堆栈可以一直增长，直到  $[SP] = [SL]$ 。硬件会阻止任何试图使堆栈超过由DB和SL所定义的界限的操作。数据区的范围是从DB寄存器中指定的位置到数据界限 (DL) 寄存器中指定的界限。

指针寄存器说明了堆栈当前的大小、最大容量和在内存中的位置。这样堆栈就是一个很容易变化的动态结构。图11-5给出了另一个堆栈指针——堆栈标志 (Q) 寄存器。这个寄存器表示

当前程序的数据堆栈的开始位置。实际上，Q指向堆栈四字入口中的第四个字，称为堆栈标志，它是为了便于在程序之间进行转换控制而设置的。Q寄存器的角色和在第2章中描述的帧指针寄存器很相似。当一个程序必须挂起时，比如产生了中断，那么正确返回到该挂起程序所需的信息将以堆栈标志的形式保存到堆栈当中。

堆栈标志的第一个字保存的是变址寄存器的当前内容，第二个字保存返回地址。实际保存的返回地址信息与PC的值及PB寄存器中的内容不同，PC中的值是指向当前程序要执行的下一条指令。由于存储的是偏移量而不是绝对地址值，所以程序可以移出这个存储器以外，之后返回到这个内存中的其他地方。通过在PB寄存器中装入新值来指向内存中的一个新区域。第三个字保存的是状态寄存器中的状态信息，第四个字保存的是该堆栈标志和它紧挨着的前一个堆栈标志的距离。

图11-6给出一个堆栈标志，表示为 $k$ ，在 $\text{Procedure}_k$ 初始化时放入堆栈；另一个堆栈标志，表示为 $k+1$ ，当 $\text{Procedure}_{k+1}$ 初始化时放入堆栈。当这个新程序完成时，机器通过堆栈标志 $k+1$ 中的数据将控制权交给先前的程序。那时，Q寄存器必须设置指向堆栈标志 $k$ 的第四个字。因为在每个标志中都存储了堆栈标志之间的距离，所以这很容易实现。同时，SP也要设置指向堆栈标志 $k+1$ 的先前程序的位置。结果，SP指向的是 $\text{Procedure}_k$ 所用的栈顶，这样，就恢复了在调用 $\text{Procedure}_{k+1}$ 时的退出状态。这种技术可以用在循环嵌套的程序中。程序之间的参数传递也是采用堆栈来完成的。

605

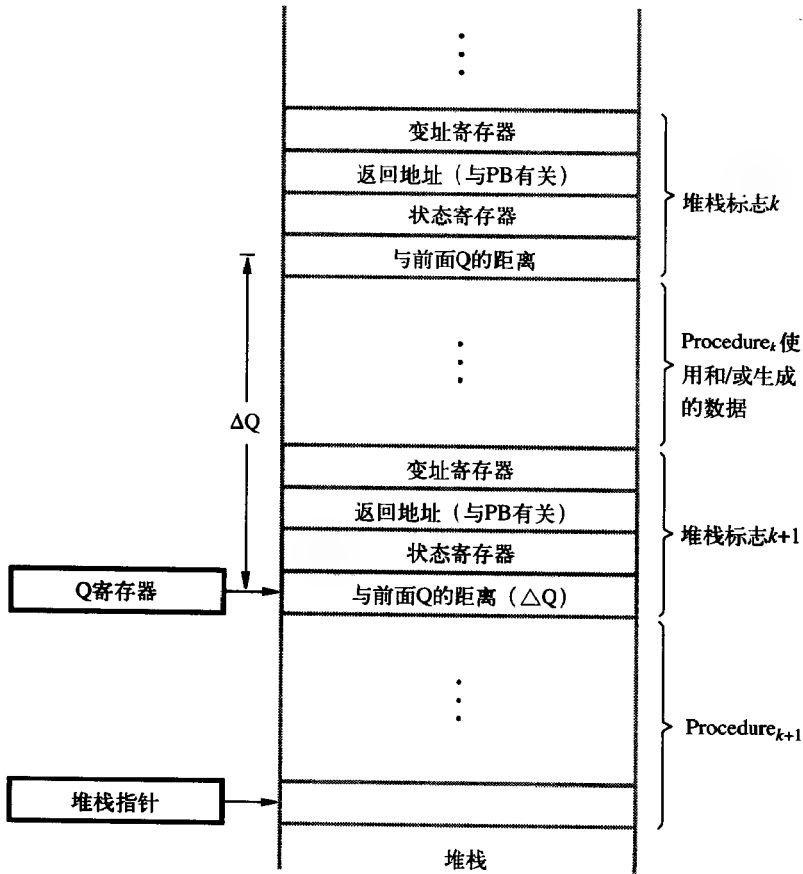


图11-6 HP3000中的堆栈标志

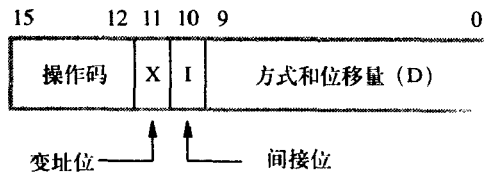
除了指针寄存器以外，HP3000计算机在机器内部结构中还使用了其他的硬件寄存器。只有两个寄存器程序员可见，即变址寄存器和状态寄存器，它们的功能本质上和大多数计算机中同名的寄存器是相同的。然而注意，这里没有给程序员使用的通用寄存器。相反，用堆栈操作的数据是临时存储的，我们在下一节中用例子进行说明。

11.8.2 堆栈指令

作为一种基本策略，堆栈计算机是在栈顶有限的空间内对数据进行操作的，而且产生的结果还要保存在堆栈中。这就要求有可以在堆栈和内存之间传送数据的指令。

HP3000有各种指令，它们都是16位。大多数的指令在某种方式上包括堆栈和典型的操作数、操作地址，或者是其他保存在堆栈中的相关参数。HP3000有13类主要指令。我们不描述整个HP3000指令集，而是把重点放在机器的堆栈结构上。首先来考虑存储器地址指令，其格式在图11-7中给出。

606



方式	位模式										有效内存地址
	$b_9$	$b_8$	$b_7$	$b_6$	$b_5$	$b_4$	$b_3$	$b_2$	$b_1$	$b_0$	
PC + relative	0	0	← D →								[PC] + D
PC - relative	0	1	← D →								[PC] - D
DB + relative	1	0	← D →								[DB] + D
Q + relative	1	1	0	← D →							[Q] - D
Q - relative	1	1	1	0	← D →						[Q] - D
SP - relative	1	1	1	1	← D →						[SP] - D

图11-7 HP3000中内存寻址指令格式

5位操作码字段的11个标量用来指定这类指令。存储器地址指令包括：

LOAD 将一个指定的内存字压栈。

STOR 将栈顶字弹出，放入指定的内存单元中。

ADDM 将指定的内存字和栈顶相加，并将栈顶改成相加的和。

MPYM 将指定的内存字和栈顶相乘，并将栈顶改为乘积中最重要的字。

INCM 将指定的内存字加1。

607

这些指令采用特定的寻址方式来指定内存中的操作数，操作数的地址放在相应的寻址方式所使用的PC、DB、Q或SP寄存器中。10位方式和偏移字段分别用来表示采用的寻址方式和相应的偏移量，如图中所示。因为偏移字段是从6位到8位可变，所以所有的寻址方式可以采用不同长度的偏移量。变址位和间接位表示是否使用了变址寻址或间接寻址。它们是图11-5中用来访问数据区操作数仅有的两种寻址方式。

第二类指令是传送指令，使用一到两个内存操作数。这些指令可以在内存中传送字（字节）、比较两个字节串或扫描一个字节串直到找到特定的值为止。内存地址可以在相应的寻址方式中重新计算。偏移量在指令中并没有明确给出，而是以数据的形式保存在堆栈中。此外，只能通过相关的程序或数据的基址，也就是PB或DB寄存器中的内容来指定地址。这类指令中的一个典型的例子是基本的MOVE指令，它在内存中将k个字从一个位置移到另一位置中，其中：

- 第一个堆栈元素TOS指定k。
- 堆栈的第二个元素给出与PB或DB相关的第一个源内存单元的地址。
- 堆栈的第三个元素给出与DB相关的第一个目标内存单元的地址。

由于该指令的大部分寻址数据和参数长度都是隐含在堆栈的指定位置上的，因此它可以表示成16位代码。但在指令执行之前这些数据必须装到堆栈中。

下面我们讨论堆栈指令，其格式由图11-8给出。这类指令采用最高位是四个0来区别。剩下的12位用来说明特定的指令，并且这12位被分成两个6位的域，每个域用来指定一个目标操作。6位长度一共可以定义64个不同的堆栈操作。这个数字已经足够容纳各种堆栈操作了。一条指令用10位（主操作码加上堆栈操作码A）来指定一个堆栈操作，剩下的6位不考虑。当用剩余的位来指定第二个堆栈操作（使用堆栈操作码B）的时候，它要在第一个操作完成后才执行。这样，两条堆栈操作可以打包成一条单指令。因为寻址数据和操作数不作为指令的一部分包括在其中，所以可以有效地利用指令代码空间。

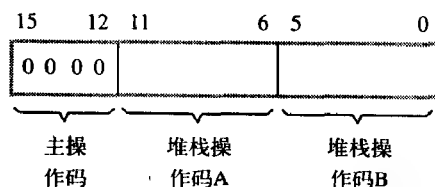


图11-8 HP3000中堆栈指令格式

一些堆栈指令实例如下：

**ADD** 将栈顶的两个字相加，并把这两个元素删除，然后将相加的和放入栈顶。

**CMP** 将栈顶的两个字进行比较，设置完相应的条件码之后将这两个字删除。

**DIV** 用栈顶整数字除堆栈的第二个整数字，用商替换第二个字，余数替换栈顶。

**DEL** 删除栈顶字。

堆栈指令集中提供了很多的指令，尽管有些指令是非常复杂的。例如，长除指令（DIVL）用栈顶元素来除以第二和第三个元素构成的双字整数。之后删除这三个元素，分别将余数和商作为栈顶元素和第二个元素放入栈中。我们在这里讨论堆栈指令时比较宽松地使用了“指令”这个术语，因为加法操作和除法操作可以在一条指令中指定。当描述这样的操作时更习惯用指令作为术语，而且这也和先前描述的将两条指令打包成一条指令的技术相对应。只有执行两个连续的堆栈操作时才能将其打包。在其他情况下，操作码B是不使用的。

迄今为止，我们只强调了压缩指令的一个优点——减少所需的代码空间。它的另一个优点是减少了内存的访问次数，因为两条指令可以作为一个16位字进行有效的读取。必须记住，在执行堆栈指令期间必须访问堆栈中的操作数。如果堆栈是在内存中，就需要进行内存的访问了。

为了说明堆栈在算术运算中的作用是用来临时保存计算的中间结果的，我们来看一个简单的例子。图11-9给出算术表达式

$$w = \frac{(a+b)}{c/d+(e \times f)/(g+h)}$$



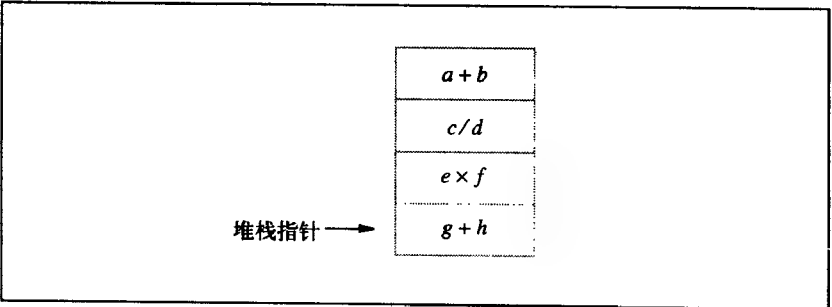
是如何进行计算的。我们假定变量 $a, b, \dots, h$ 不在栈顶。它们保存在内存地址为 $A, B, \dots, H$ 的区域中，并可以通过图11-7给出的寻址机制进行访问。此外，假定所有操作数只是单字长整数。图中给出了所需的13步操作。所需的操作是按照表达式从左到右，先分子再分母的顺序进行的。使用的符号中栈顶元素（TOS）表示为 $S$ 。这样，操作  $S \leftarrow [S] + [B]$  的含义是栈顶元素和操作数 $B$ 相加，并用相加的和代替原栈顶元素。操作  $S \leftarrow [S - 1] / [S]$  表示堆栈的第二个元素被栈顶元素除，并将两个操作数从栈中删除，再把商和余数放入栈中。

HP3000机器指令需要执行图中所示的必要计算。其功能在本节的前面已经描述过。除了除法指令以外，大多数步骤的实现采用的是单指令操作。DIV指令分别用商和余数来替换被除数和除数。因为我们只对商感兴趣，所以用DEL指令将余数从栈中删除。当遇到两个连续的堆栈指令时，就将它们合并成一个16位的指令。所有的中间结果都保存在堆栈上。图11-9b给出了在完成第9步时的栈顶元素。

609

步骤	执行的操作	机器指令
1	$S \leftarrow [A]$	LOAD A
2	$S \leftarrow [S] + [B]$	ADDM B
3	$S \leftarrow [C]$	LOAD C
4	$S \leftarrow [D]$	LOAD D
5	$S \leftarrow [S - 1] / [S]$	DIV DEL
6	$S \leftarrow [E]$	LOAD E
7	$S \leftarrow [S] \times [F]$	MPYM F
8	$S \leftarrow [G]$	LOAD G
9	$S \leftarrow [S] + [H]$	ADDM H
10	$S \leftarrow [S - 1] / [S]$	DIV DEL
11	$S \leftarrow [S - 1] + [S]$	ADD
12	$S \leftarrow [S - 1] / [S]$	DIV DEL
13	$W \leftarrow [S]$	STOR W

a) 所执行的操作和需要的机器指令



b) 在第9步之后保存在堆栈中的临时结果

610

图11-9 在处理表达式 $w = (a + b) / [c / d + (ef) / (g + h)]$ 时堆栈的使用

### 11.8.3 堆栈中的硬件寄存器

计算机中对内存单元的访问是最关键的时间制约因素之一。从内存中读取一个操作数所需要的时间要长于在处理器的寄存器中进行处理的时间。这也是处理器中要包括通用寄存器的主要原因。在堆栈计算机中，通用寄存器的临时保存功能是通过堆栈机制来实现的。如果堆栈是完全在内存中实现的，由于所有临时的保存场所是堆栈的一部分，那么处理器必须频繁地访问内存。

将整个堆栈都用硬件寄存器来实现，其成本太高且不太灵活。不过可以对全寄存器或全存储器实现堆栈方法进行一个折中，即将大部分的堆栈放在内存中，而将栈顶的几个元素保存到处理器的硬件寄存器中。这样，由于大部分的访问只涉及栈顶的几个元素并且只需在寄存器和处理器之间进行传送，所以访问堆栈的时间会大大降低。在HP3000计算机中，用四个寄存器来保存四个栈顶元素。

在堆栈中包含硬件寄存器意味着真正的栈顶元素（TOS）通常是寄存器中的一个。这就表明SP再没有必要指向一个内存区域。为了能在任何时刻都掌握栈顶元素所处的位置，通过两个寄存器来实现SP的功能。用一个16位的内存堆栈寄存器（SM）来存储堆栈当前所占内存的最高地址。用一个3位的寄存器（SR）来表示硬件寄存器中当前包含的栈顶元素的个数，其取值为：0，1，2，3或4。因此，[SP]的值是：

$$[SP] = [SM] + [SR]$$

这个值就等于将所有堆栈元素放在内存时栈顶元素所处的内存地址。它的结构在图11-10中有说明。程序员并不会觉察到堆栈中硬件寄存器的存在。对程序员而言，只存在一个指针——堆栈指针，即SP。

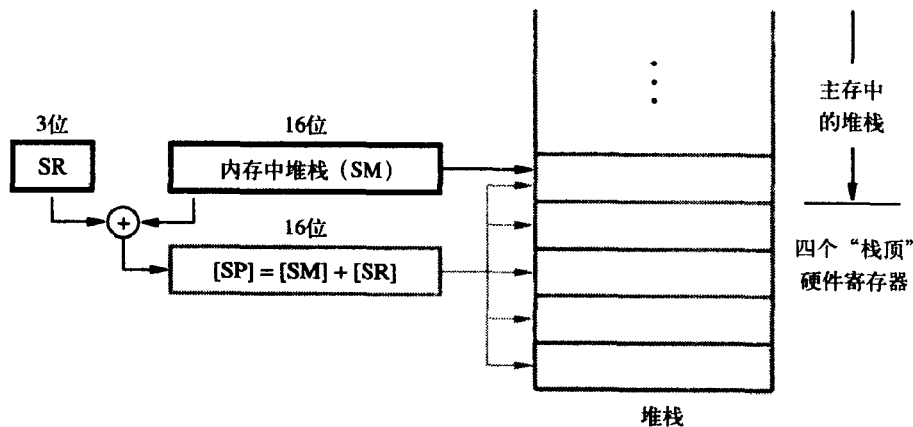


图11-10 HP3000中的栈顶结构

611

## 11.9 结束语

高性能的处理器都采用了片上的高速缓存来完成指令、数据和地址的转换操作。这些处理器都有多重独立的基于流水线的处理单元，并且在每个时钟周期内能向这些单元发出一条以上的指令，有的还可以按照乱序处理。动态转移预判和推测性轮流执行程序路径的复杂方法，使得通过判断的方式执行指令成为可能，这样就提高了计算机的总体性能。这些特点是提高程序

指令执行效率的基本方法,且与是否采用RISC或CISC设计方法来实现指令集和寻址方式无关。将高级语言程序转化成高效机器语言代码的优化编译器也是提高性能的一个关键因素。在评估一台计算机的性能时必须要考虑很多因素,正如在第1章中讨论的那样,标准化的基准程序通常是为了评估机器性能而使用的。

## 习题

- 11.1 ARM指令(参见第3章部分I)中的条件执行与IA-64指令中的预判执行之间有何关系?说明其异同点。
- 11.2 ARM指令中16位的Thumb指令子集主要用于压缩程序编码。估算一下在图11-9中所示的算术表达式需要多少条Thumb指令。假定Thumb指令集中有一条合适的除法指令(实际上并不是这样的)。与图11-9a中的HP3000程序指令数相比,Thumb指令的指令数是多少?
- 11.3 讨论Motorola 680X0处理器系列与Intel 80X86处理器系列之间的异同点。从最初版本一直到68040和80486版本之间都要进行比较。
- 11.4 在68030微处理器中有一个256字节的指令高速缓存和一个256字节的数据高速缓存。是否这样就一定比只有一个512字节的指令高速缓存而没有数据高速缓存的结构好呢?两者的优缺点各是什么?针对只有一个统一的512字节的指令和数据高速缓存回答同样的问题。
- 11.5 Intel IA-32处理器有专门用于I/O操作的指令,如同第3章部分III中描述的那样。Motorola 680X0处理器只使用存储器映射I/O,那么存储器映射I/O相对于专用I/O的优缺点是什么呢?
- 11.6 讨论Motorola 680X0和Intel 80X86处理器中相应寻址方式的优点。应特别注意在每种处理器的寻址方式中是如何便于程序重定位、堆栈的实现、访问操作数表以及字符串操作的。
- 11.7 在11.3.1节中介绍了IA-32处理器可以根据所采用的分段和分页方法,将存储器划分为四种不同的组织方式。对四种可能的方式举出一些例子来说明相应的优点。
- 11.8 写一个ARM、68000或IA-32程序来计算图11-9中的算术表达式。程序中所需的机器指令与图中给出的数值相比如何?假定标准的ARM指令集中有一条合适的除法指令(实际并不存在)。
- 11.9 在Alpha处理器中,只有32位和64位的Load和Store指令是在高速缓存和处理器之间直接使用数据通道的。描述一个组合逻辑网,该网能够利用32位宽的数据通道将任意一个四字节的数据加载到目标路径中的低位字节。
- 11.10 对IA-64处理器中的寄存器堆栈操作和第2章中处理器(内存)堆栈操作进行比较。尤其要对堆栈指针SP和IA-64模式中相对应的帧指针FP进行比较。
- 11.11 请给出执行用于管理IA-64寄存器堆栈的指令Alloc X,Y时所需硬件的一般性描述。假定使用小的寄存器和加法器。它们是如何使用的?
- 11.12 Alpha 21264处理器的高速缓存方案与21164有很大的不同。为什么21264的方案较好呢?也就是说,当只考虑缓存的作用时,什么情况下用21264执行程序的速度更快?
- 11.13 说明表达式

$$w = a \left[ (b \times c) + (d \times e) + \frac{f \times g}{h \times i} \right]$$

在HP3000中是如何计算的。

11.14 在HP3000计算机中, Procedure<sub>i</sub>产生8个字数据DI<sub>1</sub>, ..., DI<sub>8</sub>, 并存储在堆栈中。当这些字放到堆栈之后, 且在Procedure<sub>i</sub>执行之前, 一个新程序Procedure<sub>j</sub>被调用, 它产生10个字数据DJ<sub>1</sub>, ..., DJ<sub>10</sub>, 也存在堆栈中。之后, 另一个程序Procedure<sub>k</sub>被调用并在堆栈中放入3个字数据。请画出此时栈顶元素的内容。

11.15 说明表达式

$$w = (a + b)(c + d) + (d \times e)$$

分别在HP3000、ARM、Motorola 68000和IA-32计算机中是如何进行计算的。变量 $w$ ,  $a$ ,  $b$ ,  $c$ ,  $d$ 和 $e$ 的值存储在内存中。并作如下假定: 地址是不连续的。在HP3000中用DB+相关模式的直接内存寻址。在Motorola 68000和IA-32计算机中采用绝对/直接内存寻址。在ARM计算机中采用相对寻址方式。所有乘积都是单字长的。

613

11.16 在执行图11-9给出的程序时所占用的最大的堆栈数是多少?

11.17 在习题11.13和习题11.15中, 用HP3000程序重新回答习题11.16的问题。

## 参考文献

1. S. Furber, *ARM System-on-Chip Architecture*, 2nd ed., Addison-Wesley, Great Britain, 2000.
2. *IEEE Micro*, vol. 17, no. 4, eight articles on ARM, July/August 1997.
3. ARM web site: arm.com
4. D. Tabak, *Advanced Microprocessors*, McGraw-Hill, New York, 1991.
5. Motorola web site: motorola.com
6. J. Circello, et al., "The Superscalar Architecture of the MC68060," *IEEE Micro*, vol. 15, no. 2, April 1995, pp. 10-21.
7. Intel web site: intel.com
8. D. Alpert and D. Avnon, "Architecture of the Pentium Microprocessor," *IEEE Micro*, vol. 13, no. 3, June 1993, pp. 11-21.
9. R.P. Colwell and R.L. Steck, "A 0.6-Micron BiCMOS Processor with Dynamic Execution," *Proceedings of the International Solid State Circuits Conference*, February 1995.
10. "A Tour of the P6 (Pentium Pro) Microarchitecture," Intel Corporation, 1995.
11. S.K. Raman, V. Pentkovski, and J. Keshava, "Implementing Streaming SIMD Extensions on the Pentium III Processor," *IEEE Micro*, vol. 20, no. 4, July/August 2000, pp. 47-57.
12. Advanced Micro Devices web site: amd.com
13. *Communications of the ACM*, vol. 37, no. 6, eight articles on the PowerPC, June 1994.
14. *IEEE Micro*, vol. 14, no. 5, five articles on the PowerPC, October 1994.
15. IBM web site: ibm.com
16. K. Diefendorff, et al., "AltiVec Extensions to PowerPC Accelerates Media Processing," *IEEE Micro*, vol. 20, no. 2, March/April 2000, pp. 85-95.
17. M. Tremblay and J.M. O'Connor, "UltraSparc I: A Four-Issue Processor Supporting Multimedia," *IEEE Micro*, vol. 16, no. 2, April 1996, pp. 42-50.

18. Sun Microsystems web site: sun.com
19. T. Horel and G. Lauterbach, "UltraSPARC III: Designing Third Generation 64-bit Performance," *IEEE Micro*, vol. 19, no. 3, May/June 1999, pp. 73–85.
20. *Digital Technical Journal*, vol. 4, no. 4, issue on Alpha, 1992.
21. Compaq web site: compaq.com
22. E. McLellan, "The Alpha AXP Architecture and 21064 Processor," *IEEE Micro*, vol. 13, no. 3, June 1993, pp. 36–47.
23. J.H. Edmondson, et al., "Superscalar Execution in the 21164 Alpha Microprocessor," *IEEE Micro*, vol. 15, no. 2, April 1995, pp. 33–43.
24. R.E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro*, vol. 19, no. 2, March/April 1999, pp. 24–36.
25. *IEEE Micro*, vol. 20, no. 5, six articles on the IA-64 architecture and the Itanium processor, September/October 2000.
26. C. Dulong, "The IA-64 Architecture at Work," *COMPUTER*, vol. 31, no. 7, July 1998, pp. 24–32.
27. R. Krishnaiyer, et al., "An Advanced Optimizer for the IA-64 Architecture," *IEEE Micro*, vol. 20, no. 6, November/December 2000, pp. 60–68.
28. R.P. Colwell, et al., "A VLIW Architecture for a Trace Scheduling Compiler," *IEEE Transactions on Computers*, vol. 37, no. 8, August 1988, pp. 967–979.
29. D. Patterson and D. Ditzel, "The Case for the Reduced Instruction Set Computer," *ACM SIGARCH Computer Architecture News*, vol. 8, no. 6, October 1980, pp. 25–33.
30. M. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, MIT Press, Cambridge, MA, 1985.
31. W.A. Samaras, N. Cherukuri, and S. Venkataraman, "The IA-64 Itanium Processor Cartridge," *IEEE Micro*, vol. 21, no. 1, January/February 2001, pp. 82–89.

614

615  
1  
616

# 大型计算机系统

### 本章目标

在本章中你将学习以下内容:

- 由多处理器或多计算机组成的大型计算机系统
- 实现多处理器的不同结构
- 互连网络和LAN
- 多处理器中的存储器组织结构
- 共享数据的高速缓存一致性
- 共享存储器和消息传递范例
- 多处理器系统中的性能问题

617

当一个计算机应用程序需要在一定的时间内进行大量的计算时,我们必须使用具有较强计算能力的计算机。这样的机器通常称为巨型计算机。巨型计算机的典型应用包括天气预报、建筑设计中有限元分析、流体分析、大型复杂的物理系统模拟以及计算机辅助设计(CAD)。在前面章节中没有对巨型计算机进行讨论。

我们可以用快速电路技术和体系结构的特性,如多功能单元、流水线、大容量的高速缓存(cache)、交替式主存以及指令与数据总线相分离等方式设计出高性能的处理器。所有这些的可能性一直都处于探索和研究中,并且有些制造商正在努力开发用于工作站的处理器。他们追求的是在不提高成本的情况下尽可能提高机器的性能,而且已经取得了令人瞩目的成果——目前工作站的性能要胜过十年前巨型计算机的性能。

然而,很多应用所要求的计算要远远超过工作站的现有能力,对超级计算能力的要求仍然很强烈。一种方法是制造一个带有几个功能强大处理单元的巨型计算机。该类型通常是通过采用快速电路技术、更宽的通道、访问更大的内存以及扩展I/O能力来实现的。这种计算机需要相当大的能源和昂贵的散热方案。在计算型需求的应用中,巨型计算机需要尽可能高效地处理数据向量,这里向量是指一个数字(元素)的线性数组。操作通常在整个向量上执行。例如,一条加法操作可以生成一个向量,该向量是由两个64位的向量的逐个元素相加得到的。同样,一条单独存储器访问操作可以控制整个向量在主存和处理器寄存器之间进行数据传送。如果应用程序有助于向量处理,那么具有向量体系结构的计算机就会显示出良好的性能。这类巨型计算机已经由一些公司如Cray(Cray-1, Y-MP和SV1)、Fujitsu(VP5000)、Hitachi(SR8000)和NEC(SX-5)投放到市场中。此类机器的主要缺点是成本太高——无论是购买价格还是操作与维

护成本都是非常昂贵的。

一种可提供超级计算能力并且具有吸引力的方案,是在工作站中使用大量的处理器。该方案可以通过两条基本途径来实现。第一种是构造一个包含有有效高带宽媒介的机器,可以在多个处理器、存储模块和I/O设备之间进行通信。这样的机器通常指的是多处理器。第二种是通过局域网将多台工作站相互连接起来构成一个系统。这类系统通常称为分布式计算机系统。多处理器和分布式计算机系统有很多相似之处。前者具有很高的性能但价格昂贵,后者则可以用较低成本通过现代计算机环境获得。在本章的后续中,我们将讨论这些类型的显著特点。它们以合理的开销提供很强的计算能力。

618

使用多处理器构成的系统其高性能可以通过将多个计算并行处理而获得。提高这种系统使用效率的难点之一就是如何将一个应用分解成较小的任务,并把它们分配到单个处理器上同时进行处理。划分这些任务并使之能够在多个处理器中调度并协调执行,这通常需要很复杂的软件和硬件技术。我们在本章的后面讨论这些内容。

## 12.1 并行处理的形式

对于一个给定的计算任务,将其中的某些部分进行并行处理,通常有很多的处理方法。我们在前面一些章节中已见过几种。例如,在I/O操作的处理中大多数计算机中都有这样的硬件,它们可以在I/O设备和主存之间进行直接存储器访问(DMA)。在主存和磁盘之间,无论任何一个方向上的数据传送都可以在DMA控制器的控制下在处理器之间并行执行。

当将一个数据块从磁盘传送到主存时,处理器通过向DMA控制器发送指令开始传送过程。当控制器按照时钟周期发送所需的数据时,处理器可以执行一些和这些数据传输无关的计算。当控制器的传输完成时,它向处理器发送一个中断请求信号通知处理器所需的数据已经传送到主存中。相应地,处理器就可以切换到使用这些数据的计算中。

这个简单的例子说明了并行处理的两个基本特征。第一,整个任务具有这样的特点,它的一些子任务是可以在不同的硬件部件上并行完成的。在本例中,处理器计算和I/O传输可以通过处理器和DMA控制器并行执行;第二,必须有一些方法来启动和协调并行活动。当处理器设置DMA传输开始后可以继续其他的计算。当传输完成时,由DMA控制器向处理器发出中断信号以实现两者之间的动作协调,从而允许处理器开始使用传送过来的数据继续进行相应的计算。

前面的例子说明了只包括两个任务的简单并行例子。一般来说,应用程序中有大量的计算可以进行细分,从而能够进行并行处理。在计算机中有一些硬件结构是用来支持这样的并行计算的。

### 并行结构分类

619

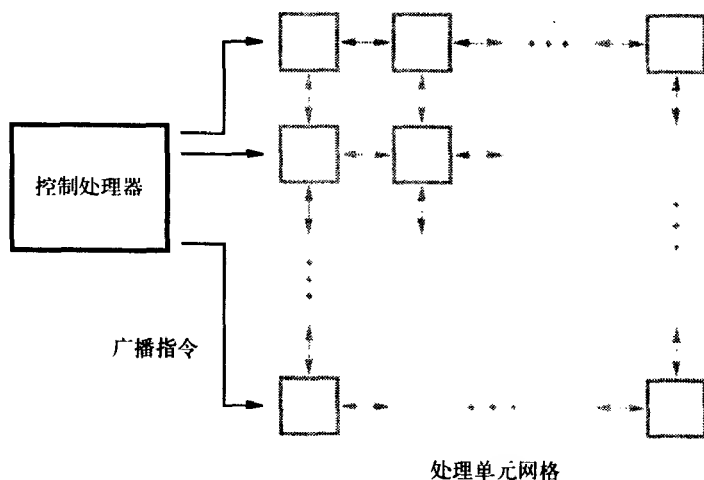
并行处理的一般性分类是由Flynn<sup>[1]</sup>提出的。在这个分类中,单处理器计算机系统称为单指令流单数据流(SISD)系统。处理器执行的程序组成单指令流,它操作的数据项序列组成单数据流。第二种模式中,单指令流广播到多个处理器中。每个处理器都有自己的数据。在这种模式中,所有的处理器都执行同样的程序但操作不同的数据,这通常称为单指令流多数据流(SIMD)系统。该系统中,多个数据流是按照数据项访问的顺序保存在每个单独处理器的存储器中。第三种模式包括多个独立的处理器,每个处理器都执行不同的程序但是访问它们自己的数据。这样的机器称为多指令流多数据流(MIMD)系统。第四种模式是多指令流单数据流

(MISD) 系统, 在这个系统中, 不同的处理器对公用的数据进行操作, 每个处理器都执行不同的程序。这种计算形式在实际中不会出现, 所以在此不做讨论。

本章主要讨论MIMD结构, 一般来讲它是最有用的。但是, 我们首先简要地浏览一下SIMD结构, 说明这种应用是如何进行良好匹配的。

## 12.2 阵列处理器

并行处理中的SIMD形式也称为阵列处理, 它是并行处理研究和实现的最初形式。在20世纪70年代早期, 一种名为ILLIAC-IV<sup>[2]</sup>的系统利用这种方法在Illinois大学设计成功, 后来由Burroughs公司制造。图12-1说明了一个阵列处理器的结构, 其中二维处理单元网格用来执行从中心控制处理器广播出的指令流。在广播每条指令时, 所有的单元同时执行。每个处理单元与四个相邻的单元相连以便进行数据交换。角上的单元与相对应的行与列连接, 在图中没有给出。



620

图12-1 阵列处理器

下面我们考虑一个特殊的计算来理解SIMD结构的计算能力。处理单元网格可以解决二维问题。例如, 如果网格的每个单元代表空间中的一点, 那么阵列可以用来计算平面区域内点的温度。假定平面边缘的温度是固定的。处理单元所表示的离散点的近似解决方法可通过以下步骤得到: 外部边缘由指定的温度来初始化。内部点是用任意值进行初始化的, 各点之间可以不同。然后反复对每个单元进行并行处理。每次迭代都是用一个点周围的四个最近相邻点的平均值来计算, 以改善该点处的温度值。当两次连续迭代之间估算值的变化量小于预先定义的精度时, 就停止这一过程。

阵列处理器的处理能力在进行这样的计算时绰绰有余。每个单元必须通过图中所示的路径与相邻点交换数据。在每个处理单元中都有包含一些存储数据的寄存器和本地内存。为了便于与相邻点之间进行数据交换, 又设置了网络寄存器。中央处理器可以广播指令将网络寄存器中的数据向上, 下, 左, 右四个方向移动一步。每个单元还包含一个ALU来执行控制处理器所广播的算术指令。利用这些基本的设置, 就可以重复广播指令序列并执行迭代循环。控制处理器必须能够检测每个处理单元的温度何时达到所要求的计算精度。为实现这个要求, 每个单元在内部设置了一个状态位, 用1来表示该条件已经成立。互连网格还包括允许控制器检测在迭代结



束时所有的状态位全部置位的装置。

对于阵列处理器来说有一个有趣的问题，就是面对少量功能强大的处理器和大量功能简单的处理器这两种情况，选择哪种效果会更好呢？ILLIAC-IV是支持前一种选择的典型例子。该处理器内部结构由64个64位处理器构成。20世纪80年代后期间世的阵列处理器是后一种选择的例子。由Thinking Machines 公司生产的CM-2机器可以容纳65 536个处理器，而每个处理器位宽只有一位。Maspar公司的MP-1216可容纳处理器的最大数量达到了16 384个，每个处理器的位宽是4 位。Cambridge Parallel Processing Gamma II Plus机器中的处理器数达到4 096个，并且可以处理8个字节长或位长的操作数。这些选择反映了这样一种思想，在SIMD环境中，高度的并行性要比少量功能强大的处理器更加适用。

阵列处理器是具有很强针对性的机器。它们特别适用于处理用矩阵或向量形式表示的数字问题。前面具有向量体系结构的超级计算机也是适合解决这类问题的。基于向量的机器与阵列处理器系统的主要区别在于，前者是通过大量的流水线技术以达到较高性能，而后者是通过复制计算模块来提供大规模的并行能力。无论是阵列处理器还是向量机都只是用于对通用计算进行提速，它们没有很大的商业市场。

621

### 12.3 通用多处理器结构

在上节中描述的阵列处理器主要是为了解决计算类问题而设计，这类计算具有明显的数据并行的特点。而对于一般的并行性不是很明显的情况，采用MIMD体系结构就显得更加有效，该体系结构中包括多个能够独立并行处理不同程序的处理器。

图12-2、图12-3和图12-4中给出了实现多处理器系统的三种可能方式。最常见的模式如图12-2所示。其中的互连网络允许 $n$ 个处理器访问 $k$ 个存储器，这样使得任何处理器可以访问任意一个存储器。但是互连网络会在处理器和存储器之间带来一定的延迟。最普遍的情况就是访问所有存储器的延迟都相同，这种机器称为统一存储器访问（Uniform Memory Access, UMA）多处理器。因为处理器执行指令所用时间极短，从存储器中取指令和数据时，网络中的延迟如果太长是无法忍受的。遗憾的是，具有很短延迟的互连网络是通过提高成本和复杂性来实现的。

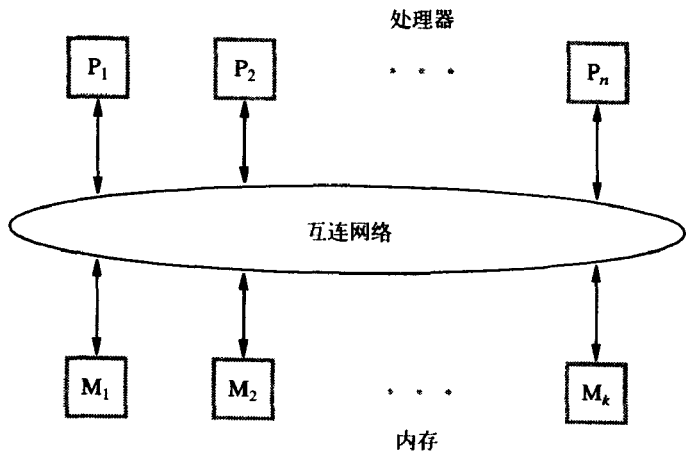


图12-2 UMA多处理器

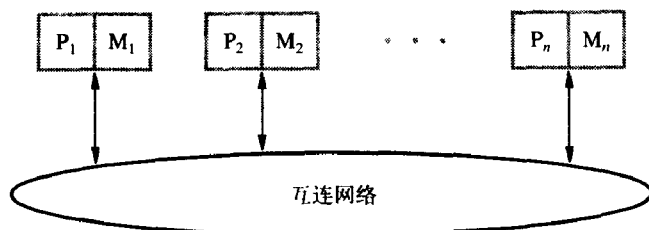


图12-3 NUMA多处理器

622

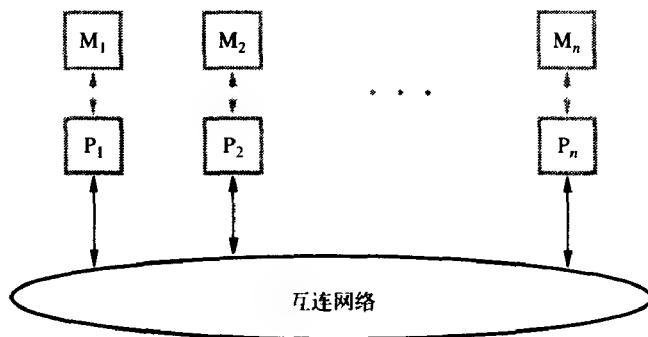


图12-4 分布式存储器系统

另一种较好的选择可以在所有处理器中保持较高的计算速度，它将存储器和处理器直接组织在一起。这种组织结构如图12-3所示。每个处理器除了可以访问自己的存储器以外，还可以通过网络访问其他存储器。由于远程访问要通过网络，这种访问相对于访问本地存储器来说时间要长一些。因为访问时间的不同，这样的多处理器称为非统一存储器访问（Non-Uniform Memory Access, NUMA）多处理器。

图12-2和图12-3所示的组织结构都提供了全局存储器，每个处理器可以访问任何存储器而无须干扰其他的处理器。在图12-4中给出了一种以不同方式组织的系统。其中，所有的存储器都是处理器的私有存储器，并且和处理器直接相连。一个处理器在没有远程处理器的协调下是不能访问远程存储器的。这种协调通过处理器之间的消息交换来实现。这样的系统通常称为带有消息传递协议的分布式存储器系统。

前面是将处理器和存储器模块作为多处理器系统的主要功能单元来讨论的。尽管我们没有明确讨论I/O模块，但是任何多处理器都必须提供大量的I/O能力。这种功能可以通过不同的方式来实现。单独的I/O模块可以利用标准接口直接连接到网络上，如同在第4章中讨论的那样。有些I/O功能还能与处理器模块做成一体。

623

图12-2、图12-3和图12-4是站在较高的层面上来认识这些可能的多处理器组织结构的。这些机器的性能和成本主要依赖于实现的细节。在以下两节中，我们将研究实现通信网络和存储器层次结构的最流行方案。

## 12.4 互连网络

在本节中，我们检验一下在多处理器系统中实现互连网络的一些可能性。通常，网络必须能够允许信息在系统中任意两个模块之间进行传递，同时还可以利用广播将信息从一个模块传

送到其他的模块。其中网络流量是由请求（例如读和写）、数据传送和各种命令组成。

某个特定网络是否适用往往是通过它的成本、带宽、有效吞吐量和实现的难易程度等术语来判断的。带宽是指相关数据的传输能力，它使用每秒传送的位数或字节数来描述。有效吞吐量是数据传输的实际速率。这个速率要小于可用的带宽，因为一个指定的连接并不是一直都在进行数据的传送，有时候该连接是处于空闲状态的。

网络传输的信息通常是固定长度和固定格式的包。例如，读请求可能是一个只包含源地址（处理器模块）和目的地址（存储器模块）以及用来指示读操作类型命令字段的信息包。向存储器模块写入一个字的写请求可能是只包含被写数据的单个信息包。相反，一个包括整个高速缓存块的读响应需要几个信息包。更长的信息还会需要更多的信息包。

理想情况下，一个完整的信息包可以在一个时钟周期内在网络中的任何节点或开关上并行处理。这就意味着要有许多缆线构成大量的连接。为了降低成本和复杂性，通常尽可能减小连接的宽度。在这种情况下，一个信息包必须分成更小的信息包，以便使每个小的信息包在一个时钟周期就可以传送到。

#### 12.4.1 信号总线

将大量的模块互相连接起来的最简单、最经济的方法是使用信号总线。与第4章所讨论的一样，总线设计的特点在这里也适用。由于几个模块是与总线连接的，因此每个模块在任何时刻都可以请求进行数据传送，这样就必须有一个高效的总线仲裁方案。我们在第4章中给出了这种方案的例子。

一种简单的操作模式就是在整个请求传送期间，总线只分配给特定的源-目标对而不进行另外的分配。例如，当处理器在总线上发出一条读指令时，它就会占用总线直到获得来自内存的所需数据为止。由于存储器模块需要一定的时间来访问数据（在第5章中描述），总线就会一直处于空闲状态直到存储器将数据准备好，然后将数据传送到处理器中。只有当这个传送结束时，总线才释放出来进行其他请求的处理。

624

假定一次总线传输用 $T$ 个时间单位，内存访问时间为 $4T$ 。然后它需要用 $6T$ 完成一次读请求。这样总线就会有 $2/3$ 的时间处于空闲状态。一种称为分割事务协议（split-transaction protocol）的方案可以使总线在空闲期间为其他请求提供服务。考虑以下处理一系列来自不同处理器读请求的处理方法问题。在第一个请求包的地址传送完成以后，总线可以重新分配给第二个请求来传送地址。假定这个请求与前面请求的是不同的内存模块，那么现在这两个模块的读访问周期可以并行进行。如果它们都没有完成访问操作，那么总线还可以继续分配给第三个请求，依次类推。最终，第一个内存模块完成了它的访问周期后，利用总线将数据传送给请求的源地址。在其他模块完成各自的周期时，也利用总线将数据传送到相应的源地址。需要注意的是地址传送和返回字之间的实际时间长度不是关键的。对于不同请求的地址和数据传送可以独立表示总线按照任意顺序交替使用的情况。

分割事务协议允许更有效地使用总线和可用的带宽。该协议所带来的性能提高主要是依赖于总线的传输时间和内存访问时间两者之间的关系，而且提高的性能是以提高总线复杂性为代价的。这里复杂度的提高有两点原因：其一，内存模块需要知道发出读请求的源地址，必须在请求中附带着请求源的标识。因为之后存储器要利用该源地址将请求的数据发送回去；其二，必须是所有模块都能够对总线进行控制，而不仅仅是处理器具备该功能。

对于使用分割事务总线的多处理器系统，其处理器数的范围是4个到32个。如果规模再增加，总线的带宽就会成为问题。更宽的总线具有更宽的带宽，因此也就要用更多的缆线。大多数在处理器和内存块之间传送的数据都是由高速缓存块构成，每个块中有几个字。如果总线宽度足以每次传输几个字，那么，整个块的传输会比一次传输一个字要快得多。由Silicon Graphic 公司开发的具有挑战性的多处理器计算机中所采用的是允许并行传输256位数据的总线。

然而总线的主要限制是可以连接到总线的模块数量不能过高。如果一条总线上连接的模块数不超过10或15个，它会很正常地工作。利用更宽的总线来增加带宽可以使连接的模块数加倍。但总线的带宽受到连接使用的总线的限制，同时由于连接了很多的模块而带来加电引起的传输延迟相应增加，这样也会限制带宽。所以，允许多个独立传输操作且用于并行处理的网络可以大大提高数据的传输速度。

#### 12.4.2 纵横 (Crossbar) 网络

图12-5中给出了多功能交换装置。这就是著名的纵横开关，它最初是为电话网而开发的。为了便于说明，我们将图中的开关描述成机械开关，但它们实际上是电子开关。任一模块 $Q_i$ 可以通过适当的开关与任一个开关 $Q_j$ 相连。这种所有节点之间都直接连接的网络称为全连接网。在全连接网中许多传送可以同时进行。如果 $n$ 个出发点向 $n$ 个不同的目的地发送数据，那么这些传送可以并发执行。因为传送不会由于缺少通信路径而被阻塞，所以crossbar称为无阻塞交换。

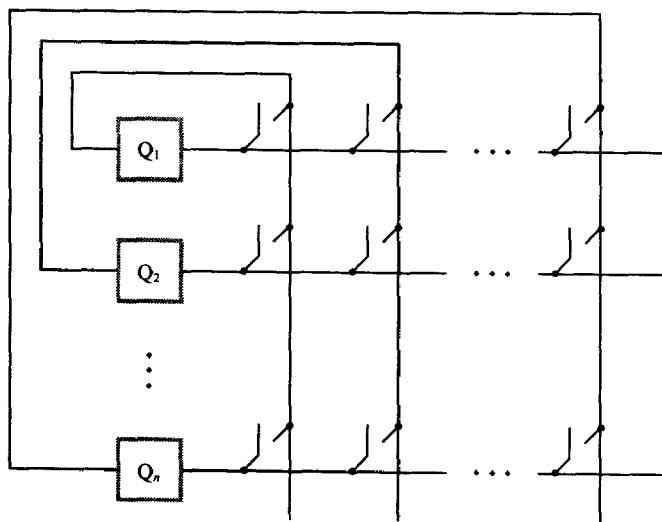


图12-5 纵横互连网络

在图12-5中我们看到在每个交叉点处只有一个开关。然而，在实际的多处理器系统中，通过纵横网的路径是非常宽的。这就意味着在每个交叉点处需要设置很多的开关。由于将 $n$ 个模块相互连接起来需要交叉点的数量是 $n^2$ ，那么随着 $n$ 的增长开关的总数会变得非常庞大。其结果是成本很高并且实现复杂。纵横网的最大优势就是当节点的数量不是很大时可提供良好的网络互连能力。

在Sun公司的E10000系统中实现了一个较大的纵横开关，它采用 $16 \times 16$ 的格式，连接16个4处理器的节点。该系统中使用了多级的纵横开关，其中第一级纵横开关连接到第二级的纵横开

关上,依次类推。按照这种方式可以连接更多的处理器。这种方法还被Fujitsu公司的VPP5000、Hitachi公司的SR8000和NEC公司的SX-5等机器所采用。对于高性能的互连媒质来说,这种多级的纵横结构已经成为一种流行趋势。

### 12.4.3 多段网络

刚刚描述的总线和纵横系统只用一段开关在源地址到目的地址之间提供路径。此外,还可以利用多段开关来实现互连,即在源地址和目的地址之间建立起多条路径。这样的网络相对于纵横结构来说成本更低,但却能够在源地址和目的地址之间提供相当多的并行路径。最好通过一个实例来说明。图12-6中给出了一个称为混洗网(Shuffle network)的三段网络,它互连了八个模块。其中术语“混洗”描述从一个段输出到下一段输入的连接方式。这种方式与玩纸牌中重新混洗的方法相同,也就是将纸牌分成两半相互交叉进行混合。

图中的每个开关盒是一个 $2 \times 2$ 的开关,这样可以对输入和输出进行路由。如果输入请求的是不同的输出,那么它们可以同时按照直接通过或交叉方式进行路由。如果有两个输入请求相同的输出,那只能满足其中一个,另一个将被阻塞直到前一个请求使用完开关后才可以执行。可以看出,一个由 $s$ 段构成的网络可以连接 $2^s$ 个模块。在本例中从网络的任意模块 $Q_i$ 到其他任意模块 $Q_j$ 只有一条通路。因此,该网络在源和目的地之间提供的是全连接。但是,当有很多请求模式时是不能同时满足的。例如,从 $Q_0$ 到 $Q_4$ 与 $Q_1$ 到 $Q_5$ 之间的连接就不能同时提供。

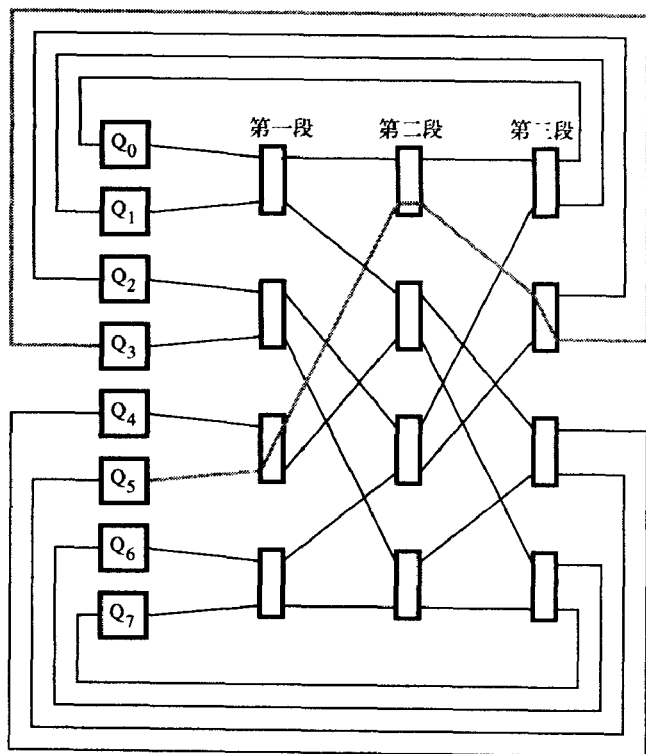


图12-6 多段混洗网

多段网络与纵横网相比,实现起来成本更低一些。如果用图12-6中的方式来连接 $n$ 个节点,

那么必须用  $s = \log_2 n$  个段, 每段有  $n/2$  个开关盒。由于每个开关盒中包含4个开关, 那么开关的总数量是

$$4 \times \frac{n}{2} \times \log_2 n = 2n \times \log_2 n$$

对于较大的网络, 显然比在纵横网中用  $n^2$  个开关要少很多。

利用下面的方法可以将一个特殊的请求通过这种网络进行路由。源地址向网络中发送一个用二进制形式表示的目标地址数。当这个数在网络中传送时, 每段都检测一个不同位来决定开关的设置。第一段使用最高位, 第二段使用中间位, 第三段使用最低位。当有请求到达开关的输入时, 如果控制位是0, 就从高位输出, 如果控制位是1, 就从低位输出。例如, 一个请求从源  $Q_5$  到目的地  $Q_3$ , 它通过网络所走的路径在图12-6中用粗线画出。它的路由是根据目标地址的位串011来进行控制的。

基于多段网络的多处理器有一个更好的例子, 是由BBN Advanced Computers 公司制造的BBN Butterfly。它是一个64处理器模型, 其中包含一个用  $4 \times 4$  开关构建的三段网络。通过每段开关的路由是利用目标地址中连续的两位来控制的。最近的例子则是IBM RS/6000 SP多处理器, 该系统可使用多段网络(选件之一)连接处理器群。

多段网络与纵横网相比, 其所提供的并发连接能力要弱一些, 但是它的实现成本却很低。这种网络方案在20世纪80年代迅速流行并达到巅峰, 但是在之后的几年很快就冷了下来。在本章后续部分所讨论的方案会更具有吸引力。

#### 12.4.4 超立方体网络

在前面讨论的三种方案中, 互连网络为连接任意的两个模块提供通路, 同时也带来了时间延迟。这种方案可以用于实现UMA多处理器。我们现在在讨论只适用于NUMA多处理器的网络拓扑。第一种流行方案使用的是  $n$  维立方体的拓扑结构, 称为超立方体结构, 它实现了一个连接  $2^n$  个节点的网络。除了通信电路外, 每个节点通常包括一个处理器和一个具有一定I/O能力的存储器模块。

图12-7给出的是3维的超立方体。其中, 小圆圈代表一个节点的通信电路。图中没有给出每个节点相连的功能单元。立方体的边代表相邻节点之间的双向通信连接。在  $n$  维超立方体中, 每个节点直接与  $n$  个相邻节点相连。这里采用为节点分配二进制地址的方式来标注, 这样任何两个相邻节点的地址之间只有一位不同, 如图所示。

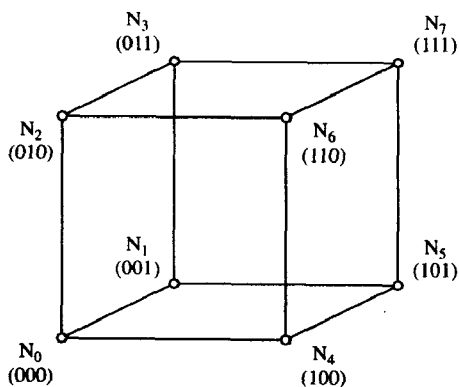


图12-7 3维超立方体网络

通过超立方体的路由消息相当简单。如果节点  $N_i$  上的处理器要向节点  $N_j$  发送消息, 将按照如下方式进行处理: 将二进制形式的源节点  $i$  和目标节点  $j$  从低位向高位进行比较。假定第一个不同的位是  $p$ 。则节点  $N_i$  向它的相邻节点(地址是  $k$ , 且与  $i$  在  $p$  位不同)发送消息。节点  $N_k$  继续利用同样的比较方法将消息发送到相应的相邻节点。这样, 消息会随着从一个节点到另一节点的一次次跳跃而逐渐接近目标节点  $N_j$ 。例如, 从节点  $N_2$  到节点  $N_5$  的消息需要经过节点  $N_3$  和节点  $N_1$  的3步跳跃。在一个  $n$  维的超立方体中, 任何消息需走的最大距离是  $n$  步跳跃。

从右向左扫描地址的方法仅仅是用来决定消息路由方法中的一种。还可以采用其他如每步跳跃都能将消息向目标点传送的方法,只要在路径上的每个节点可以通过当前的信息做出路由决策就可以。超立方体的优点是它的可靠性。两个节点之间存在多条路径就意味着当存在连接冲突时,通过简单的局部路由选择就很容易将其避免。如果最短的路径不能使用,那么消息可能要通过更长一些的路径来发送。这时必须要注意避免形成环,如果一旦形成一个封闭的环,消息就处在环内而永远不能到达目标点了。

629

超立方体互连网络已经被很多机器所采用。比较有名的例子包括Intel的iPSC,它使用了连接128个节点的7维立方体,还有NCUBE的NCUBE/ten,使用了连接1024个节点的10维立方体。在20世纪90年代早期,由于更具吸引力的基于网格结构网络的出现,而使超立方体网络不再流行。

#### 12.4.5 网状网络

将大量节点连接起来的最自然连接方式就是通过网格(mesh)。在图12-8中给出了16个节点的网状网的例子。节点之间的连接是双向的。网状网是在20世纪90年代流行起来的,已经在本质上取代了超立方体网,成为大量多处理器互连网络的主要选择。

网状网络中的路由可以有几种不同的方式。其中最简单也是最有效的一种就是在源节点 $N_i$ 和目标节点 $N_j$ 之间选择一条路径,传输按照水平方向从 $N_i$ 到 $N_j$ 进行。当到达 $N_j$ 所在的列时,传输沿着这个列在垂直方向进行。基于网状网络多处理器的几个知名的实例是: Intel的Paragon、斯坦福大学的实验机Dash<sup>[3]</sup>与Flash<sup>[4]</sup>以及MIT的Alewife<sup>[5]</sup>。

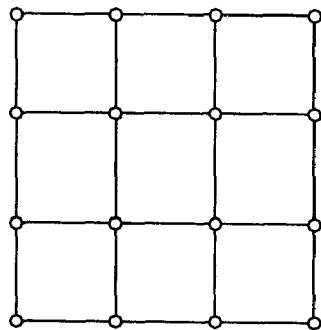


图12-8 一个2维的网状网

如果将图12-8中对边的节点相互连接,结果这个网络就变成了在X方向和Y方向上都是双向环的网。这种网通常称为圆环形(torus)网络,它可以减小信息传输的平均延迟,但它是以提升复杂性为代价的。Fujitsu公司的AP3000使用的就是这种网络。

规则的网状和圆环形都能实现3维网络。相邻节点在X、Y和Z三个方向上进行连接。在Cray T3E多处理器中就有3维圆环形的例子。

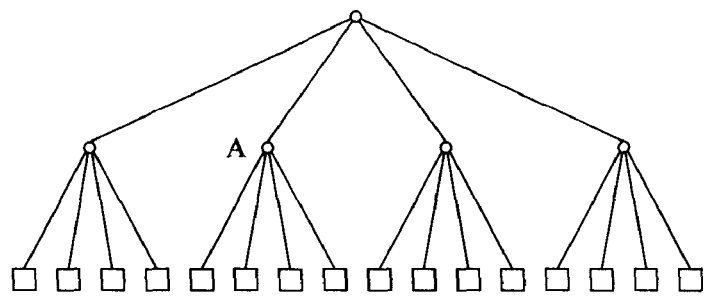
#### 12.4.6 树状网络

按照树的形状实现的网络层次结构是另外一种互连拓扑结构。图12-9a给出了连接16个模块的4路树结构。树中每个父节点一次可以与两个子节点进行通信。中间节点,如图中的节点A,可以提供从一个子节点到该中间节点的父节点的一个连接。这就使得任意距离的两个叶节点能够相互通信。在树中的一个给定的节点上一次只能建立一条通路。

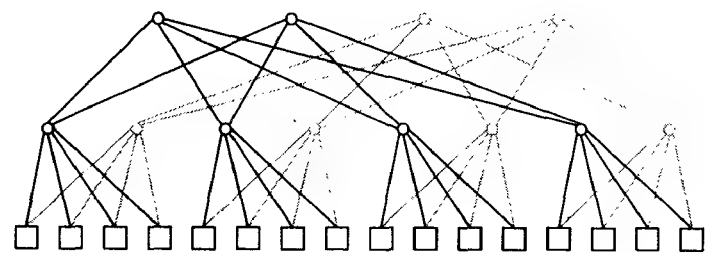
630

如果大量的通信都是局部的,也就是只有一小部分网络通信经由单一的根节点进行,那么树状网络会运行良好。如果不是这种情况,根节点将会成为瓶颈,从而使网络的整体性能迅速下降。

为了降低出现瓶颈的可能性,可以增加在树状层次结构中较高层的连接数目。因此胖树网络应运而生,树中每个节点(除了最顶层的根节点外)都有一个以上的父节点。图12-9b中给出了一个胖树的例子。这个例子中每个节点都有两个父节点。胖树结构在Thinking Machines公司的CM-5机器中所采用。



a) 4路树

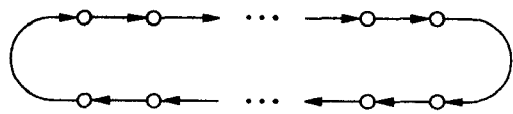


b) 胖树

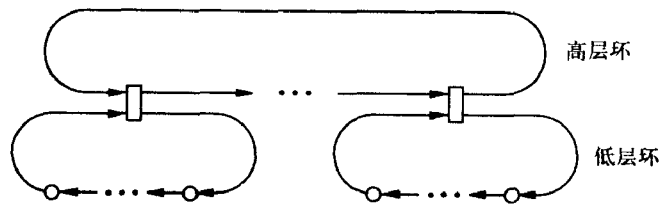
图12-9 树状网络

12.4.7 环状网络

最简单的网络拓扑之一是使用一个环将系统中的节点连接起来，如图12-10a所示。这种结构的主要优点是环很容易实现。由于每个节点只与两个相邻点连接，所以环中的连接可以很宽，通常能容纳并行处理中的一个完整的信息包。然而，将很多节点连接成一个非常长的环是没有用的，因为信息的传输延迟会太长，以至于无法接受。 631



a) 单环



b) 分层环

图12-10 环状互连网络



环还可以作为构建块用在前面所讨论的拓扑结构中,如网状、超立方体、树以及胖树。考虑一个环用在树状结构中的简单例子,一个用环形成的分层结构如图12-10b所示。图中描述了两层的环状结构,还可以有更多层。在包括相同节点数的环上,短环可以大大地减少传输延迟。此外,在不同环的两个节点之间的传输延迟会比在单环中小。但是这种方案的缺点是:最顶层的环可能会成为传输的瓶颈。

具有环状网络特征的商用机器包括Hewlett-Packard生产的样机V2600和Kendal Square Research研制的KSR-2。在多伦多大学的实验机Hector<sup>[6]</sup>和NUMachine<sup>[7]</sup>中也采用了环状网络。

#### 12.4.8 实用性因素

632 我们已经了解了在多处理器系统中实现互连网络的几种不同的拓扑结构。很难说哪种拓扑结构明显好于其他的结构。每种结构都有特定的优缺点。当对不同的方法做比较时,必须将几个实用性因素考虑进去。

最基本的要求就是通信网络要足够快,并且要具有足够的吞吐量来满足多处理器系统中的传输需求。这就意味着在通信路径上要有很高的传输速度,并且要有一个简单的路由机制来保证能快速地做出路由决策。网络必须容易实现,配线的复杂度必须是合理的并且能够简单打包。复杂度是网络成本的必然反映,也是要考虑的另外一个重要因素。

多处理器的规模是不同的。理想的网络应该适用于所有规模,规模的范围可以从只有几个处理器到有成千上万个处理器。术语可伸缩性经常用来描述多处理器体系结构(包括互连网络)随着系统规模的增加,性能也可以增加的能力。如果一个比较小的多处理器系统可以在低成本下构成,但是它能够很容易地扩展成一个庞大的系统,并且其价格和性能是按照线性增长的,那么这个系统的优势就很明显了。通常,前期的一个小系统成本也是很大的,因为一个大系统中所需的大多数通信硬件在小系统中也必须提供。

除了在源地址和目标地址之间进行基本通信以外,将一个消息传送到整个网络可以被所有节点接收的广播能力也很有用。而只给网络节点中的一个子节点发送消息的能力也是很有益的,这种传输称为多播。

互连网络的选择通常会影响到所选方案的实现,这些方案用来保证在每个不同处理器的高速缓存中保存共享数据的副本,在需要更新时及时更新,以便所有副本的值都相同。这样的方案我们将在12.6.2节中讨论。

可靠性是另外一个重要因素。越复杂的网络就越难保证其可靠性。理想情况是机器在网络连接中有一些故障时也能继续运转。这就要求在一对通信节点之间至少要存在两条不同的通路。通常情况下,简单的网络注重稳健性,它的故障率通常比系统中的处理器和内存模块低很多。包括有额外硬件的高可靠性网络的构建成本是相当高的。这个话题已经超出了本书的范围。

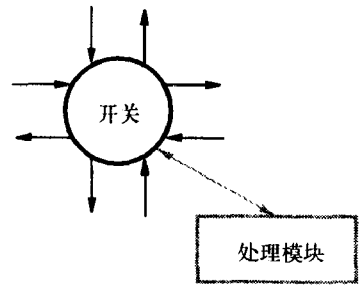
为了说明怎样评估这些因素,让我们基于网状网络和环状网络来做一个简要的定性比较。

##### 网状和环状

网状网络和环状网络的特点都是采用可以在高速时钟频率驱动下的点对点连接(连接相邻点)。两者都具有结构简单并且容易扩展的特性。在环上增加扩展相对于网状来说要更简单一些。

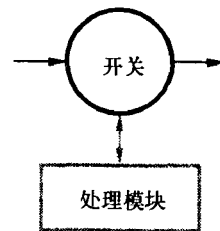
633 在图12-8和图12-10中,我们将网络中的节点表示成小圆圈并用单线连接。下面看一个更详细的图。图12-11中显示的是与带有一个处理器模块的节点相连的通信路径。开关块中包括为传输选择路径的电路和用来缓存正在传输数据的缓冲器。每个时钟周期内,数据从一个节点的缓

冲器传送到下一节点的缓冲器。图12-11a描绘了2维网状网络中的一个节点。由于在X方向和Y方向上都需要双向通信, 所以一个节点必须有八个不同的网络连接。从成本和封装的因素考虑, 其连接宽度受到使用缆线总数量的限制。这样, 单个的连接不可能宽到足以并行传送整个信息包的程度。为了解决这些限制, 可以将一个信息包分割成相应于连接宽度的较小部分进行处理。术语flit (流控制数) 通常是指一个信息包的一部分, 这个部分可以被节点中的开关电路所接受并向前传送, 或者是在向前的通道中被其他传输阻塞而进行缓冲。实际上, 为了方便起见, 通常将flit的宽度和连接宽度置成相同大小。



a) 网状网络中的节点

如果一个信息包必须分成flit, 那么该如何将它通过网络进行路由呢? 一种直接的方法是称为存储-转发的方式, 它在每个节点处都提供了大量的缓冲器用来保存信息包的所有flit。这样一个完整的信息包就可以从一个节点传送到另一个节点, 并在每个节点中存储直到可以传送到下一节点为止 (传输所需的时钟周期数是由flit的数量决定的)。这种方法的负面影响是所需的缓冲器大小以及增加了通过一个节点的时间延迟。一个更好的选择是采用虫孔 (wormhole) 路由策略 (可以参看流水线部分), 它可以将构成信息包的一系列flit看成是穿过网络的一条虫子。虫子中的第一个flit包含了带有目标节点地址的头部。当这个flit穿过网络时, 它就建立了一条通路, 其余的flit可以沿着这条通路进行传输。虫子的尾部关闭所建立的通路。因为其他虫子也许会经过同一个节点, 所以虫子的头部可能是任何节点的临时块。然而, 一旦头部开始移动, 那么虫子的其余部分会在后续的时钟周期跟着移动。当虫子的头部被阻塞时, 必须有控制机制来停止那些来自前面节点的flit的传送。一种简单的方法是每个节点将两个缓冲器分别用于两个传输方向<sup>[8]</sup>。虫孔路由与存储-转发路由相比, 其传输延迟更小, 因为它头部flit的发送无须等待该包中其余的flit。



b) 环状网络中的节点

图12-11 网状网络和环状网络中的节点

634

虫孔路由是线路交换策略的一种应用, 该策略是电话网中的常见概念, 当呼叫方拨打一个电话号码时就建立一条通过网络的路径。通话进行的路径称为线路。当呼叫方挂起电话时就解除了该线路。在虫孔路由中是通过头部flit建立通路的。这个flit前进的过程也许会被临时阻塞。一旦建立起一条线路, 信息包的其余flit就向目标移动, 而无须任何的竞争。相反, 整个信息包在每个节点进行缓存的策略, 如存储-转发的方法, 这称为包交换。这种情况无须建立线路, 只要每个节点有可用的缓冲器就可以将信息包通过网络进行传送。

图12-11b中给出了一个环状网络中的节点连接。这里除了连有处理模块以外, 只进行单方向的传输。这样, 连接的宽度就是同样数量缆线的网状网络的四倍。这也就意味着可以将整个信息包在一个时钟周期内从一个节点并行地传送到另一节点。图12-11b中给出了在环的最低层的一个连有处理模块的节点。如果使用的是图12-10b中的环状分层结构, 那么, 在低层与高层之间的内环接口将有两个输入连接和两个输出连接, 高层环和低层环中每种连接的接口将各占一个。

分层结构的环状网络中的路由是非常简单的。除了在高层网和低层网的输入信息包都要同时连续使用环中同一个内环接口的情况下会出现阻塞以外, 其他情况下信息包是永远不会被阻

塞的。为了处理这种情况，必须在接口处提供缓冲器（队列），一个用于从低层环到高层环，另一个用于相反方向。只要上游的相邻点没有信息包到达该节点时，处理模块就可以向环上输入一个新的信息包。

635 下面要看一下网络广播和多播数据的能力。这种能力是环状网络天生就具备的。例如，可以通过将一个信息包发送到最顶层环的方式来广播给所有的节点。当这个信息包穿过这个环时，在每个内环接口处都拷贝一份并将其继续发送给低层环。这一过程在所有层中重复进行，以便最初的信息包可以访问最底层的所有节点。网状网中的广播相对来说就困难一些，因为广播包要分成flit，并且广播过程中的虫子可能在不同的节点被其他传输所阻塞。此外，广播的完成不容易检测。

分层环状网络的主要缺点是，环的最顶层可能会因为过多的信息包通过而成为瓶颈。当局部通信很少时就会发生这种情况。顶层环的有限带宽限制了基于该网系统的可伸缩性，所以只能达到几百个处理器。相反，基于网状的系统可以连接上千个处理器。

从前面讨论中可以看出网状和环状都是实现互连网络的不错选择。基于环状的系统容易实现，但它的扩展性不如基于网状的系统好。如果系统的最大规模是几百个处理器的话，那么环状网的优势是很明显的。网状网在小系统和非常大的系统中也很适用。对于非常小的系统，比如有16个处理器，最有效的选择是单总线或纵横网络。

由于多处理器系统的规模具有重要的意义，读者也许想知道实际中使用的是什么范围的系统。大多数多处理器系统相对较小。很多机器有4到128个处理器。虽然也有上千个处理器的大型机，但是这种市场需求却是很小的。

#### 12.4.9 混合拓扑网络

我们已经讨论了几种可能的网络拓扑结构，并展示了这些拓扑结构所具有的优缺点。多处理器系统的设计者努力实现在合理的开销下能获得较高的性能。在开发不同拓扑所具有的最有力特点的同时，很多成功的机器采用了混合的拓扑结构。总线和纵横网络在连接少量的处理器时是极好的选择。所以，我们通常会看到利用一条总线或一个纵横方向连接2到8个处理器的处理器群。这样的群通常作为一个节点，然后采用合适的拓扑来相互连接形成一个更大的系统。

Data General公司的AV25000系统使用总线相连的处理器作为节点，然后这些节点再用环状网连接起来。Hewlett-Packard的样机V2600也是采用环状网连接节点，每个节点用一个纵横开关连接处理器。康柏的AlphaServer SC使用胖树连接节点，构成节点的处理器用纵横开关连接。

#### 12.4.10 对称式多处理器

636 假如一个多处理器系统中，所有处理器访问所有的内存模块和I/O设备的机会都是相同的，这样操作系统软件可以将任意一个处理器与其他处理器进行交换。那么，如果任意一个处理器都能执行操作系统的内核或用户程序，这种机器称为对称式多处理器（SMP）。也就是说任何一个处理器都能初始化任何I/O设备的I/O操作，以及能够处理外部的中断。

SMP通常是采用总线或纵横网来实现的。它经常在超大规模多处理器系统中当作一个节点使用。例如，上面所说的样机V2600和AlphaServer SC多处理器中的节点就是SMP。

## 12.5 多处理器的存储器组织结构

在第5章中我们看到单处理器系统中存储器的组织结构对性能有重大影响。多处理器系统中也是同样。为了利用引用局部性，每个处理器通常都带有一个主高速缓存和一个辅助高速缓存。如果采用图12-2中所示的机制，那么每个处理器模块将连接到通信网上，如图12-12所示。由于假设主高速缓存是处理器芯片的一部分，所以图中只显示了辅助高速缓存。存储器模块的访问采用的是惟一的全局地址空间，每个存储器模块分配有一个确定的物理地址范围。在这种共享存储器系统中，处理器以同样的方式来访问所有的存储器模块。从软件的角度来看，这样使用地址空间是最简单的方式。

在NUMA结构的多处理器中，如图12-3所示，每个节点包含一个处理器和一部分存储器。实现节点最自然的一种方式在图12-13中予以了说明。在本例中，还是采用全局惟一的地址空间。同样，处理器以相同的方式访问所有的存储器。但是，访问全局地址空间中的本地存储器模块所用时间要少于对远程存储器模块的访问时间。

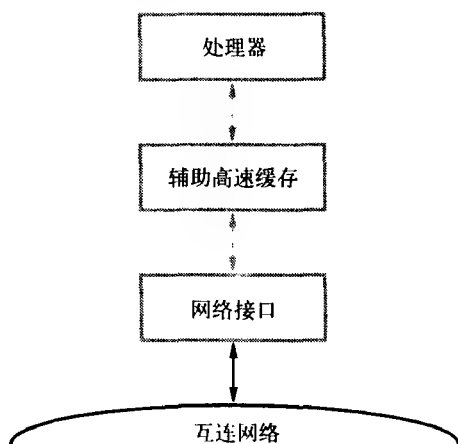


图12-12 图12-2多处理器中的一个处理器节点

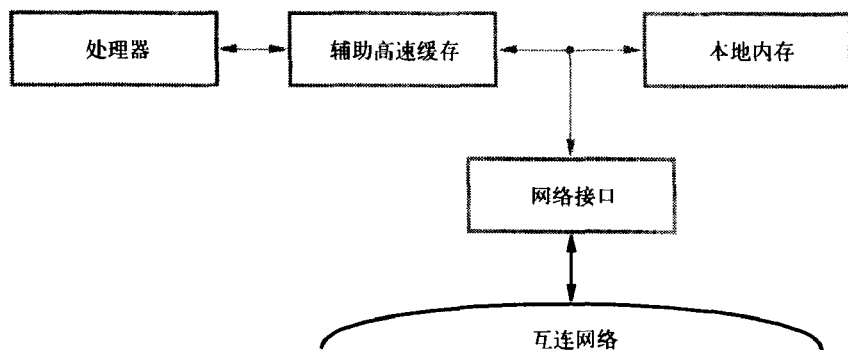


图12-13 图12-3中多处理器节点结构

图12-4的结构中，每个处理器只可以直接访问它自己本地的存储器。这样，每个存储器模块为处理器建立了私有地址空间，而没有全局地址空间。任何不同处理器中运行的程序和进程之间的交互都是通过处理器之间发送消息来实现的。在这种通信方式中，每个处理器将相互连接的网络看成是一个I/O设备。实际上，在这样的系统中每个节点都可以看作单处理器计算机。由于这个原因，这类系统也被称为多计算机系统。这种组织结构为能将一些计算机连成一个较大系统提供了最简便的方式。由于信息交换需要软件的干预，所以运行在不同计算机中的任务之间的通信相对来说要慢一些。我们将在12.7节讨论这类系统。

当在很多处理器之间共享数据时，必须保证对于一个给定的数据项，所有处理器所看到的值是相同的。在一个共享存储器系统中存在有许多高速缓存，因此在这一点上出现了问题。由于在不同的高速缓存中可能存在一些数据项的多个副本，只要一个处理器改变（写）自己高速缓存中的数据项，那么同样的操作必须在持有该副本的所有高速缓存中进行一遍。另一种选择

是其他的副本必须无效。换句话说，系统中的所有高速缓存的共享数据必须保持一致。保持高速缓存一致性的问题有几种不同的解决方案。我们将在12.6.2节中讨论最流行的方案。

## 12.6 程序并行性与共享变量

在本章的引言中阐述过，将一个大的任务分成子任务并在多处理器上并行执行是很困难的。但是，在一些比较特殊的情况下，这种分割还是容易的。如果一个  
 大任务源自于单独程序的集合，那这些子任务可以很方便地在不同的处理器上执行。除非因竞争公用的I/O设备而被阻塞，否则这样的工作方式会充分利用多处理器。

另一种情况是，当高级程序设计语言有允许应用程序员明确说明在程序中  
 可以建立并行执行的子任务这一结构时，这种情况就很容易处理了。图12-14中给出了这样的结构，我们通常把这种结构称为PAR段。这里采用控制语句PARBEGIN和PAREND将可以并行执行的部分Proc1到ProcK括起来。这个程序的执行顺序如下：当PARBEGIN语句之前的程序段完成以后，K个并行程序根据当前可用的处理器数目，来决定执行其中的一个程序乃至全部的程序。这些程序可以按照任何一种顺序开始。程序中PAREND之后部分只有在全部的K个程序处理完之后才可以开始执行。

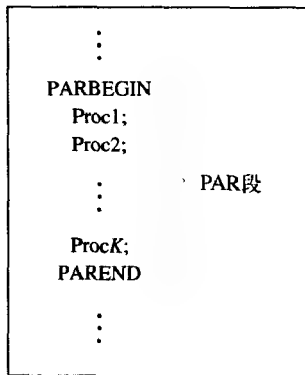


图12-14 并行程序结构

如果在多处理器中只有一个程序在执行，那么处理器的使用效率就取决于应用程序员了。PAR段的并行度K以及相对于后续段的总长度决定了多处理器利用级别。

为了提高多处理器系统利用率，最具有挑战性的任务是开发一个可以在用户程序中自动检测并行部分的编译器。自动检测并行部分的作用是基于以下原因的。应用程序设计人员很自然的想法是将一个程序看成是一个串行操作的集合。然而，即使程序员指定的操作是用某种高级语言编写的串行指令序列，但仍可能存在一些可以并行执行的不同指令。一个简单的例子是连续的循环操作。如果在循环的每次迭代之间没有数据的依赖关系，那么可以并行执行这些连续的操作。相反，如果第一遍循环中产生的数据是第二遍需要的，依次类推，那么就不能进行并行处理。编译器必须检测出数据依赖关系并决定哪些操作是可以并行执行的，哪些是不能并行执行的。这种编译器的设计是非常复杂的。即使程序中的并行部分已经明确，在处理器数量有限的多处理器系统中对其调度执行也是很艰巨的任务。调度可以由编译器来完成，也可以由操作系统在运行时进行处理。我们不深究并行执行中的决策和调度任务的执行过程，而只研究在多处理器系统中不同处理器并行运行时，程序是怎样来修改正在访问的共享变量的。

### 12.6.1 共享变量访问

假定已经识别出在多处理器上可以并行运行的两个任务。这两个任务基本上是独立的，但是它们会时常访问和修改一些共享的全局内存变量。例如，共享变量SUM代表一个账目的余额。此外再假设运行在几个不同处理器上的任务需要修改这个账目。每个任务按照如下方式对SUM进行操作：读取SUM的当前值，根据这个值执行一个操作，将结果写回到SUM中。如果分别在处理器P1和P2中运行的两个任务T1和T2执行访问SUM的读-修改-写操作的话，很容易看到错误是怎样产生的。假定T1和T2从SUM读取当前值，比如是17，然后继续在本本地修改。T1加5结果

是22, T2减7结果是10。然后将它们的结果写回到SUM, T2先写, T1后写。现在变量SUM的值是22, 哪个错了呢? SUM的值应该是15 ( $=17+5-7$ ), 在按照严格的一前一后顺序下, 修改后的哪个值是有意义的呢?

为了保证对共享变量SUM进行正确的操作, 每个任务在完成读-修改-写序列的操作期间必须是互斥访问的。这可以采用全局锁变量LOCK和一个称作测试-置位 (Test-and-Set) 的指令来实现。变量LOCK有两个值0或1。它是用来确保在执行修改该共享变量所需的时间内, 一次只能有一个任务访问SUM。这样的指令序列称作临界区。LOCK操作如下: 当没有任务进入操作SUM的临界区时, 它的值是0。当任何一个任务想要修改SUM时, 它首先检查LOCK的值, 然后不管初始值是什么都把它置1。如果初始值是0, 那么任务可以很安全地对SUM进行处理, 因为当前没有其他任务进行同样的操作。相反, 如果LOCK的初始值是1, 那么该任务就知道其他任务正在对SUM进行操作。在它能够继续处理该值以前, 必须等待直到那个任务重新设置LOCK为0。这种想得到对LOCK操作权的模式可以通过测试-置位指令很容易地实现。正如它的名字所暗示的一样, 这条指令就像单一的指令, 作为一条不可分割的操作序列来执行测试和置位LOCK的关键步骤。当这条指令执行时, 所涉及的内存模块不能响应其他处理器的访问请求。

640

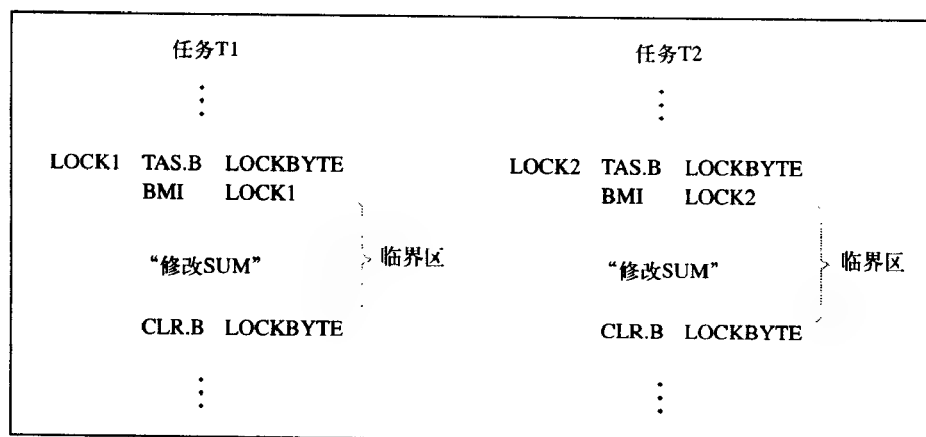


图12-15 互斥访问临界区

来看一个特殊的例子, 在Motorola 68000微处理器中将测试-置位指令表示成TAS。该指令带有一个单字节的操作数, 假定存储在内存中的LOCKBYTE处, 操作数的最高位 $b_7$ 作为刚刚讨论过的锁变量LOCK使用。TAS指令对 $b_7$ 位进行不可中断的测试和置位操作。 $b_7$ 的初始值设置成N (负值), N是条件码标志。这样, 在执行完TAS以后, 如果N等于0程序可以继续进入到临界区, 如果N等于1就必须等待。图12-15给出了两个任务T1和T2是怎样操作LOCKBYTE进入修改共享变量SUM的临界区代码的。TAS指令后面跟着条件转移指令。如果 $N=1$ , 该指令就会激活回到TAS的转移, 并在操作数的LOCKBYTE处重复执行一个等待循环, 直到TAS发现 $b_7$ 等于0为止。如果执行TAS时 $b_7$ 是0, 那么转移指令失效, 于是允许程序继续进入临界区。当临界区执行完毕, 清除LOCKBYTE。这样 $b_7$ 被重新复位成0, 于是允许任何等待的程序继续进入临界区。

TAS指令是一个可以实现锁的简单机器指令的例子。大多数的计算机中都包含这类指令。这些指令还可以提供额外的功能, 例如将检测的结果与一个条件转移结合起来。

### 12.6.2 高速缓存一致性

虽然多处理器中的共享数据会给我们的计算带来方便,但是同时也会带来另外一个问题:具有多个高速缓存就意味着在几个高速缓存中保存着共享数据的副本。当任何一个处理器在本地的高速缓存中对共享的变量进行写操作时,那么所有包含该变量副本的高速缓存中的值就会变成不正确的值。所以这一更改必须通知相关的高速缓存,这些副本要么修改为新值,要么变为无效。高速缓存一致性是对这种情况的定义,即所有在高速缓存中缓存的共享数据在任何时刻都应保持相同。

[641]

在第5章中我们讨论了两种向高速缓存中进行写操作的实现方法。直接写处理是对高速缓存和主存同时修改。写回处理只改变高速缓存中的数据,主存中的副本只是在高速缓存中的脏数据块必须要替换时才进行修改。在多处理器系统中也采用类似的方法。

#### 直接写协议

直接写协议的实现有两种基本版本。一种是修改其他高速缓存中的数值,另一种是使其他高速缓存中的副本无效。

我们先来看一下带有更新的直接写协议。当处理器向高速缓存中写入一个新值时,这个新值也写到保存该高速缓存值的内存块中。由于这个块在其他高速缓存中也存在,那么这些副本也要同时进行修改以反映出由该写操作所引起的变化。从概念上来讲,最简单的方法是向系统中的所有处理器广播要写的数据。当每个处理器接到这个广播数据时,如果在它的高速缓存(主或从高速缓存)中存在这个受影响的高速缓存块,那么就修改它的内容。

直接写协议的第二个版本是使副本无效。当一个处理器向它的高速缓存中写入一个新值时,这个值被写到内存中,并且所有其他高速缓存中的值都被无效化。然后采用广播的方式向系统中发送无效化请求。

#### 写回协议

在写回协议中,如果不同的处理器将一个高速缓存块中的内容加载(读)到其他的高速缓存中,那么这个高速缓存块就存在着多个副本。如果某个处理器想要改变这个块,它必须首先要成为这个块的独占拥有者。当保存这个内存模块的块拥有权授予该处理器时,那么所有其他的副本,包括在内存模块中的副本都被无效化。现在这个块的拥有者就可以不做额外操作而任意修改块的内容了。当有其他的处理器想要读取这个块时,就由当前的拥有者将数据传送过去。那个获得包含最新值块的拥有权和修改权的所有者,也将这些数据传送到主存模块中。

[642]

写回协议比直接写协议发生的通信量少,因为在这个块被别的处理器使用之前,该处理器执行了若干次向高速缓存块写的操作。

到目前为止,我们一直假定这些协议中的修改和无效化请求都是通过互连网络来广播的。实现这样的广播是否可行,主要取决于所采用的互连网络的结构。在12.4.1节中讨论过的单总线内部结构特点就具备这种支持广播能力的网络。在使用单总线的小型多处理器系统中,高速缓存一致性是利用监听方法来实现的。

#### 监听高速缓存

在单总线系统中,处理器和存储模块之间的全部交互是通过总线完成的。实际上,它们是向连接在总线上的所有单元进行广播。假设每个与处理器相连的高速缓存都有一个控制电路,该电路用来观察在这条连有其他处理器的总线上进行的各种交互动作。这里假定采用刚刚描述

过的写回协议。

无论何时一个处理器向它的高速缓存块进行第一次写入,该高速缓存块被标注为脏,并且将写操作在总线上进行广播。内存模块和所有其他高速缓存将它们副本变成无效。于是,执行写操作的处理器现在就成为该高速缓存块的拥有者。它可以继续对这个块进行写操作而不用将其广播。当其他的处理器发出对这个块的读请求时,因为内存模块中的副本是无效的,所以内存不能做出响应提供需要的数据。但是当这个请求出现在总线上时,该数据块的当前所有者也能看到,这时所有者必须向请求的处理器提供正确的值。存储器模块得知一个所有者正在通过广播信号来传送(包括所有者在总线上的地址数据)正确值,这时存储器模块也将自身的值一同修改。最后,所有者将它的副本标注为干净。此时,在多个高速缓存块和存储器模块中的数据都是正确值,于是操作可以继续进行。在这种情况下,为了给新数据腾出空间,必须将脏数据替换掉,所以必须对存储器模块执行写回操作。

如果两个处理器要同时向相同的一个高速缓存块中进行写操作,其中之一将被授权先使用总线并成为所有者。另一个处理器中的高速缓存块的副本就会被无效化。但是第二个处理器可以重复自己的写请求。这样,它的连续写请求可以保证在两个处理器之间进行通信,以此来保证特定高速缓存块中不同的数据统一成正确的数据。

上面描述的方法是基于高速缓存控制器能够监视总线上的动作,并可以采取相应的控制措施的能力。我们称这种方法为监听高速缓存技术。

出于对性能的考虑,监听功能不受处理器及其高速缓存正常操作的干扰是非常重要的。对于总线上的每个请求,如果高速缓存控制器都要访问其高速缓存的标志位来确定所请求的块是否在它的高速缓存中,便会产生这种干扰。如果当大部分的检测块都不在它的高速缓存中时,这种干扰是不必要的。为了消除这种不必要的干扰,每个高速缓存提供了一套与高速缓存中块信息状态相同的标志信息,用这些标志可以探听电路上单独的访问。

虽然监听高速缓存的概念很有效并且容易实现,但是它只适用于单总线系统。在大规模的多处理器系统中,必须采用更加复杂的结构。

#### 基于目录的方案

采用广播机制来发送无效化和修改请求以加强高速缓存一致性的方法,随着多处理器系统规模的扩大而逐渐显得力不从心。其主要原因是由于完全的广播所带来的大量不必要的信息传输,因为在实际的应用中,某个特定块的副本通常只存在于几个高速缓存中。

非常有用的做法是保存一个位置目录,即可以了解任何时刻副本所在的高速缓存位置。实现这种方案的一种方式是在一个特定的内存块中为每个块设置额外的状态位,用来指向这个块的副本在哪些高速缓存中。然后,不是采用前面的方法对所有高速缓存进行广播,而是内存模块仅仅向具有副本的高速缓存发送单独的消息,或多播一个写回协议的无效化请求。当然,内存块中的附加位增加了这些模块的成本。目前,已经提出了不同的目录方案版本,并且其中一些在现存的多处理器系统中已经得到实现。

#### SCI标准

电气和电子工程师协会(IEEE)已经对高速缓存一致性的详细实现方法制定了标准。该标准是SCI(可扩展一致性接口)标准<sup>[9]</sup>中的一部分,它定义了多处理器底板,该底板具备的特点是:可以提供快速的信号、可升级的体系结构、高速缓存一致性并且实现简单。互连网络采用点对点连接,其通信协议采用单个请求单个响应的规则。源节点生成的一个信息包只发往一个



目标地址。如果目标节点接收到源节点的信息包,那么它会发回一个肯定的应答信息包。如果没有收到这个信息包,就会发回一个否定的应答信息包,将该信息包重新发送。

高速缓存一致性是基于分布式目录协议实现的。包含共享数据的每个高速缓存块都有一个对应的双链表。每个处理器节点可以缓存共享数据块,以及包括指向共享块的前一节点和后一节点的指针。这些指针是高速缓存块标记的一部分。这个双链表的表头指向共享数据块所在的内存块。当有一个新节点访问内存来读取这个块时,该节点就成为链表的新表头,同时要改变内存目录,用新表头的地址来代替先前的表头。只有表头才可以对内存进行写访问。如果有其他节点要执行写操作,可以将该节点本身作为表头插入表中,并重新调整表中的其他表项。

SCI 的高速缓存一致性方案由于所需的内存目录和处理器高速缓存标志存储不会随着链表规模的增大而增加,所以具有很好的性能。这种方案的一个缺点是在任何情况下,都会有这种固定的额外存储开销,所以代价比较高。

尽管SCI标准并没有为互连网络指明一种特定的拓扑结构,但环状拓扑结构是最自然的选择之一。Hewlett-Packard的样机V2600和Data General的AV25000多处理器系统就采用了环状拓扑结构以及上面描述的一致性协议。

### CC-NUMA 多处理器

644 在多处理器系统中的高速缓存一致性是一个重要的问题。它已经成为广泛研究的课题。我们已经简要描述了一些关键的实现方案。本书不再对其细节进行更多的讨论。

多处理器可以通过硬件和软件的方法来实现高速缓存一致性。从性能上来看,用硬件来控制高速缓存一致性更有优势。目前大多数NUMA多处理器是采用硬件的方式来实现高速缓存一致性的,通常称为“高速缓存一致的NUMA (CC-NUMA) 系统”。

### 12.6.3 加锁和高速缓存一致性

我们应该注意到用锁来保护共享变量的访问是与高速缓存一致性的控制相独立的,并且这两类控制都是必需的。考虑一下这种情况,采用直接写策略的同时还修改共享变量的方式来保证高速缓存的一致性。假定12.6.1节例子中的SUM的内容已经读到正在执行T1和T2任务的两个处理器的高速缓存中。如果读操作是修改序列中的一部分,并且没有用锁保护控制进行互斥控制,那么仍然会产生最初的错误。如果像以前一样,任务T1最后写它的新值,那么SUM的值将是22,仍是错值。在整个动作期间一直都保持着高速缓存一致性,却因为没有采用锁保护控制而得到错误的结果。

## 12.7 多计算机

在12.5节中我们介绍了多计算机系统的概念。现在详细讨论该系统的显著特征。

多计算机系统的结构如图12-4所示。系统中的每个处理节点是一个独立的计算机,它们可以通过在网络中发送消息而在彼此之间通信。相对应于前面讨论的共享存储器多处理器系统而言,这类系统通常称为消息传递系统。

在多计算机系统中,互连网络上的命令不像在共享存储器多处理器系统中那样迫切。共享存储器系统中因为处理器模块要频繁访问远程共享内存模块,所以必须具有很高带宽的快速网络。如果采用慢速的网络,那么网络本身很快就会变成瓶颈而使系统的总体性能迅速下降。

在多计算机系统中,消息的发送频率远远低于共享存储器多处理器系统中的发送频率,所以系统中的信息流量会远远小于后者。因此可以使用更加简单和成本更低的网络。从不同的通

信强度来看,分别用术语紧密耦合与松散耦合来描述共享存储器系统和消息传递系统。

12.4节中描述的任何网络都可以用于多计算机系统中。由于需要的通信量相对来说比较适度,所以互连网络的物理实现可能会便宜一些。网络中的连接经常包括由I/O设备接口驱动的位串行线。接口电路通过DMA技术从源计算机的内存中读取一个消息,将它转换成位串格式,并通过网络将其传送到目标计算机中。源地址和目标地址中都包含用作路由的信息头部。这个信息被路由到目标计算机后,通过该机的I/O接口将其写到内存缓冲区中。

[645]

基于超立方体的互连网络在20世纪80年代非常流行。该网络采用的是信息传递系统,经常使用位串进行传输。这种机器的例子有Intel的iPSC、NCUBE的NCUBE/ten以及Thinking Machines的CM-2。之后在20世纪90年代早期,其他用于信息传递和共享存储器系统的拓扑结构逐渐流行起来。Thinking Machines的CM-5是使用连接宽度为4的胖树网络的消息传递机器。Intel的Paragon采用了连接宽度为16的网状网络。为了方便消息的传递,在网络中的每个节点处使用特殊的通信单元。例如,在Paragon机器中有一个消息处理器,它实质上使得应用处理器从消息处理的细节中解放出来。

### 12.7.1 局域网

因为在多计算机系统通信要求相对来说低一些,我们可以考虑用一些容易获取的标准网络来代替专门的互连网络,这些标准网络是为了更一般的通信目的而开发的。很多网络由于可以连接各种不同类型的计算机设备而存在。地理范围较小且一般距离不超过几公里的网络称为局域网(LAN)。覆盖面积较大且距离超过几千公里的网络称为远程网或广域网。

最流行的LAN采用总线或环状拓扑结构。这两种局域网的传输介质可以是双绞线、同轴电缆或光纤。它采用位串方式传输,速率范围从每秒十兆到几百兆字节。在单条共享通路上每次只能传输一个信息包。信息包中,源和目的设备地址在数据字段之前,相应的分隔符用来指示信息包的开始和结束。通常,信息包的长度是可变的,范围从几十字节到上千字节。

实现分布式访问控制的协议需要保证在任意的两个通信设备之间进行有序的传送。我们将描述两种广泛采用的协议——以太网总线和令牌环。这些协议在IEEE标准<sup>[10]</sup>中有详细的说明。

### 12.7.2 以太网(CSMA/CD)总线

以太网总线访问协议也称作带冲突检测的载波监听多路访问(CSMA/CD)协议,是在概念上最简单的协议之一。每当一个相连的设备要发送消息时,它会一直等待直到总线空闲的时候才开始传输。然后该设备在它传送消息的 $2\tau$ 秒后监视总线,其中的 $\tau$ 是数据在总线上从一端到另一端的传输延迟。如果该设备在 $2\tau$ 间隔内没有发现自己传输信号的任何失真,那么就认为没有其他的设备进行传输,可以继续传输直到结束。相反,如果发现了由其他设备开始传输而引起的失真,那么两个设备都停止传输。两个传输信号之间的相互破坏所引起的失真称为冲突,时间间隔 $2\tau$ 称为冲突窗口。

[646]

被冲突破坏的消息必须重传。如果冲突中所包括的设备立即进行重传,那么它们的信息包很可能再次冲突。防止重复冲突所采用的基本策略如下:每个单独的设备等待一个随机的时间,然后直到总线空闲才开始传输。如果随机等待时间是 $2\tau$ 的倍数,那么重复冲突的可能性就会降低。

### 12.7.3 令牌环

令牌环协议用于环状网中。一个单独的、进行适当编码的短消息称为令牌，它在环上连续不断地循环传送。令牌到达环上的哪个节点，就表示允许哪个节点进行传输。如果该节点没有任何数据传输，它要尽快地将令牌传送到下游的节点。如果该节点有要发送的数据，那么它就不传送令牌。相反它会传送一个带有适当头部编码标志的信息包。这个信息包在环上传输时，当经过目的节点的时候，它的内容就被目的节点读取并复制过去。然后这个信息包还继续在环上传输，直到回到源节点之后被丢弃。当源节点完成了一个信息包的传输之后，它就释放令牌让其在环上继续循环传递。令牌环上的信息包大小是可变的，因为目的节点必须能保存整个信息包，所以只受每个节点可用缓冲区空间的限制。

在多计算机系统的上下文中考虑标准LAN的主要原因不是因为它们是独立系统，而是因为可以利用这些网络很方便地将标准的工作站连接起来，形成多计算机系统。

### 12.7.4 工作站网络

如今，大多数商业、教育和政府机构为了满足计算的需要都建有工作站。这些工作站通常连接到LAN上，并可以访问文件服务器、打印机和专门的计算资源（参见图12-16）。

647

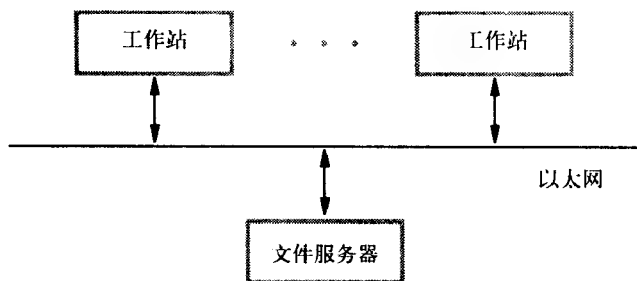


图12-16 典型工作站网络

虽然每台工作站通常是作为单独的计算机来使用，但很多工作站可以看作是一个多计算机系统。所需要的就是进行并行处理的软件。当然，这样的系统和商用的信息传递多处理器机器之间还存在一些重要的区别，尤其是通过LAN来进行通信的速度很慢。主要区别是在不同计算机上运行的程序之间进行信息交换时，必须由操作系统进行干预。这就是说工作站网络不能像带有指定互连网络的独立系统那样进行工作。但是最大优点是这种工作站网络通常可以随意获得。当然，当工作站不是作为常规目的来使用时，可以在这样的系统中运行很大的应用程序，一般这种工作在晚上进行。

## 12.8 共享存储器和消息传递实例

在前面几节中，我们考虑的是具有共享存储器和消息传递特征的多处理器系统的硬件含义。现在来简要说明一下这些特征是如何影响用户（实现并行应用程序的程序员）的。考虑一个只有两个处理器的简单例子。这样可以简化讨论并且能够说明主要问题。

假定要计算两个N元素向量的点积。图12-17给出了这个任务的串程序。该程序很适于在单处理器上运行。程序中的大部分是自解释的。Read说明将两个向量从磁盘（或其他的I/O设备）

载入内存中。这个任务由操作系统来完成。显然，潜在的并行性在于循环部分，该循环先产生两个元素的乘积，然后将其结果与先前累计部分的点积相加。

### 12.8.1 共享存储器实例

如图12-18所示，我们先试图向两个处理器中写入一个程序。当程序在一个处理器中开始执行时，它将向量装入内存并将变量 `dot_product` 初始化为0。我们通过第二个处理器执行其中一半的计算并产生所需的点积。它通过生成单独的线程在另一个处理器上运行。

```
integer array a[1..N], b[1..N]
integer dot_product
.
.
read a[1..N] from vector_a
read b[1..N] from vector_b
dot_product := 0
do_dot (a, b)
print dot_product
.
do_dot (integer array x[1..N], integer array y[1..N])
  for k:= 1 to N
    dot_product := dot_product + x[k] * y[k]
  end
end
```

648

图12-17 计算点积的串程序

```
shared integer array a[1..N], b[1..N]
shared integer dot_product
shared lock dot_product_lock
shared barrier done
.
.
read a[1..N] from vector_a
read b[1..N] from vector_b
dot_product := 0
create_thread (do_dot, a, b)
do_dot (a, b)
print dot_product
.
do_dot (integer array x[1..N], integer array y[1..N])
  private integer id
  id := mypid()
  for k:= (id*N/2) + 1 to (id + 1)*N/2
    lock (dot_product_lock)
    dot_product := dot_product + x[k] * y[k]
    unlock (dot_product_lock)
  end
  barrier (done)
end
```

图12-18 在共享存储器机器的两个处理器上计算点积程序的首次尝试

649

线程是程序内部的一条独立的执行路径。实际上，这里的线程是指执行程序的多线程中的

控制线程，这些线程就像是单独的程序一样可以并行运行。这样，在不同的处理器上可以运行两个或两个以上的线程，并且这些线程可以执行相同或不同的代码。关键一点是所有的线程都是单个程序中的一部分，并且运行在相同的地址空间上。我们应该注意到在平常的单处理器环境下，每个程序只有一个控制线程。

图12-18中的程序通过`create_thread`语句来生成一个新的线程。该线程在执行`do_dot`程序之后就结束。操作系统将为新线程分配一个标识号1。第一个处理器作为线程0继续执行`do_dot (a, b)`语句。语句`id := mypid()`用分配的线程标识号来设置变量`id`。在`for`循环中利用`id`值可以简化说明向量`a`和向量`b`哪个应该用特殊的线程进行管理。

由于不断改变`dot_product`变量的累加值在`do_dot`程序中是临界区，因此，每个线程必须互斥访问这个变量。这可以用12.6.1节中讨论的加锁机制来实现。线程0不能通过`do_dot`程序中的屏蔽语句，直到其他线程也到达该同步点。这就保证了在线程0输出最终结果之前，两个线程都已经完成了。屏蔽的概念可以通过不同的方式来实现。最简单的方法是利用一个共享变量，正如图12-18中的一样。该变量初始化为线程数（在本例中是2），之后在每个线程到达屏蔽点时增1。

图12-18中的程序有一个主要的缺点。所采用的加锁机制并没有实现所期望的并行操作，因为两个线程可以向相同的共享变量`dot_product`进行连续的写操作。这样，所需计算中潜在的并行部分实际上是串行处理的。

为了达到所需的并行，我们可以像图12-19中那样进行修改。不是采用共享变量`dot_product`，而是在`for`循环中采用私有变量`local_dot_product`，在每个线程执行期间来累计点积。这样，只是在循环结束时才需要更新共享变量，即进入`dot_product`的临界区。这样修改就允许两个线程并行执行`for`循环。

650

```

shared integer array a[1..N], b[1..N]
shared integer dot_product
shared lock dot_product_lock
shared barrier done
.
.
read a[1..N] from vector_a
read b[1..N] from vector_b
dot_product := 0
create_thread (do_dot, a, b)
do_dot (a, b)
print dot_product
.
.
do_dot (integer array x[1..N], integer array y[1..N])
  private integer local_dot_product
  private integer id
  id := mypid()
  local_dot_product := 0
  for k:= (id*N/2) + 1 to (id + 1)*N/2
    local_dot_product := local_dot_product + x[k] * y[k]
  end
  lock (dot_product_lock)
  dot_product := dot_product + local_dot_product
  unlock (dot_product_lock)
  barrier (done)
end

```

图12-19 在共享存储器机器的两个处理器上计算点积的高效率程序

这个例子很容易扩展并使用更多的处理器。需要做的只是增加线程数量即可。For循环的循环条件表达式决定基于分配的id值的计算中每个线程使用的元素范围。

图12-19中程序的效果依赖于数据向量的大小。向量越大，这种方法的效果就越好。对于小的向量，创建线程和提供同步的总开销往往会超过并行处理所带来的好处。

### 12.8.2 消息传递实例

在这个例子中内存是分布式的，每个处理器都只能直接访问自己的内存。所需的程序将在两个处理器上运行，并且明确地将数组分成两半，每一半将分别存储在每一个处理器的内存中。程序的每个副本将只访问它自己的数据。这类应用称为单程序多数据（SPMD）。读者应该注意到这类应用和12.1.1节中介绍的SIMD类型之间的不同。SIMD类型中，所有的处理器在任何时刻都执行相同的指令。

图12-20中给出了一个可能的程序。向量数据必须首先载入两个处理器的专有内存中。id值为0的程序在操作系统的协助下从磁盘上读取向量a的前一半，并将数据以相同的名字存储在它的内存中。然后它读取向量a剩余的一半，并将其放到称为“temparray”的内存缓冲区中。下一步它就向id值为1的执行该程序的处理器发送缓冲区数据的信息。向量b中的数据也重复同样的操作。id值为1的程序接收到向量a和向量b的后一半数据，然后将它们以相同的名字存储在各自的内存中。

651

```
integer array a[1..N/2], b[1..N/2], temparray[1..N/2]
integer dot_product
integer id
integer temp
.
.
id := mypid()
if (id = 0) then
    read a[1..N/2] from vector_a
    read temparray[1..N/2] from vector_a
    send (temparray[1..N/2], 1)
    read b[1..N/2] from vector_b
    read temparray[1..N/2] from vector_b
    send (temparray[1..N/2], 1)
else
    receive (a[1..N/2], 0)
    receive (b[1..N/2], 0)
end
dot_product := 0
do_dot (a, b)
if (id = 1) send (dot_product, 0)
else
    receive (temp, 1)
    dot_product := dot_product + temp
    print dot_product
end
.
.
do_dot (integer array x[1..N/2], integer array y[1..N/2])
    for k:= 1 to N/2
        dot_product := dot_product + x[k] * y[k]
    end
end
```

图12-20 在两个处理器上计算点积的消息传递程序

现在do-dot程序只是计算N/2元素的点积。注意循环的条件对于两个处理器来说是相同的, 因为每个处理器使用的数据都存在于自己的内存中。消息传递通过处理器完成do\_dot程序的执行时所采取的动作来说明: id值为0的程序将计算和显示最终的点积。当该程序接收到由程序1发来的带有部分点积值的消息时, 就完成这两步操作。程序0将接收到的这个值放到称为temp的缓冲区中。

此外, 很容易看出该例子也能扩充并使用更多的处理器。这时, 向量将分成相应的份数以便分配给每个处理器进行计算。处理器之一, 例如执行id=0的程序的处理器, 被指定用其他处理器发送过来的数据计算最终结果。

在多处理器上建立并行处理的总开销由将程序载入不同处理器所需的时间、根据不同的处理器相关的内存中划分出相应的数组所用的时间, 以及在处理器之间传送其他信息所用的时间构成。它的性能优势依赖于向量的大小和所使用的处理器数。

共享存储器和消息传递范例都有一定的优缺点。共享存储器环境使用起来更自然, 因为它是单一处理器程序模式的一个扩展, 可以更容易编写出高效的并程序。如果数据驻留在远程存储器模块中, 访问延迟可能会很明显, 最好把对全局变量的写访问降到最小。网络的通信量很可能会致使网络成为瓶颈。过程同步是程序员的任务, 它会对应用的性能产生很大的影响。

由于私有内存中地址空间的不同, 消息传递所提供的编程环境不是很自然。消息传递的时间开销是非常重要的, 因此程序员必须尽力使程序具有良好的结构, 以便减小时间开销。由于消息传递的频繁度不高, 互连网络不可能成为问题。在处理器之间的消息传递中隐含着同步操作。也许消息传递的最大优势就在于更加廉价以及硬件更容易获得这两个方面。

## 12.9 性能因素

本章的重点放在为了降低运行一个大的应用所需的时间而采用的多处理器系统的设计。性能度量的最重要方式是可获得的加速, 即对一个多处理器系统和单处理器运行同一个应用所用的时间进行比较。加速的定义是

$$S_p = \frac{T_1}{T_p}$$

其中 $T_1$ 和 $T_p$ 分别表示使用一个处理器和使用 $P$ 个处理器时所需的时间。图12-21中给出了当将系统处理器数看作一个函数时, 可能出现的三类加速情况。直观上来讲, 我们希望随着处理器数量的增加, 并行处理一个应用所需的时间应该相应地减少。这应该是一个线性加速, 即 $S = P$ , 这也是可扩缩系统的目标。不幸的是这个目标实现起来并不容易。

正如上一节讨论的, 一个应用程序中不可能每个部分都可以并行处理的。程序中串行执行的部分所用的时间是相同的, 与使用的处理器数没有关系。正是这些串行执行的部分限制了相应的加速部分。

无法达到线性加速的另一原因是由初始化、同步、通信、高速缓存一致性以及加载不平衡等因素所带来的过高的系统开销。系统开销往往会增加系统的规模。我们在前面章节中遇到的系统开销的例子中不包括加载不平衡。通常必须等待最后一个处理器完成之后才处理下一批任务。因此, 当一个并行任务分布在多个处理器中的时候, 如果所有处理器同时到达给定同步点时效率是最高的, 这时候加载也是平衡的。

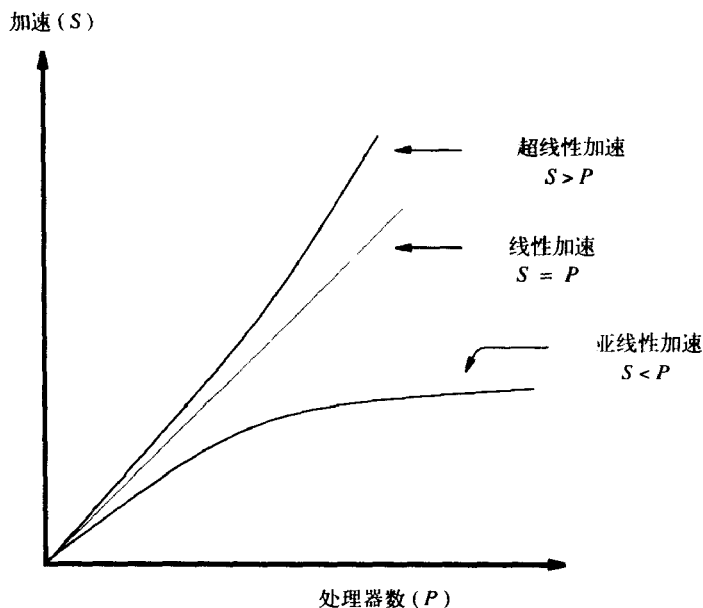


图12-21 多处理器系统的加速曲线

实际系统中大多数应用可获得的加速是亚线性的，并且处理器的个数达到一定数量时，性能便不再提高了。有一些奇怪的现象，在一些应用中的加速是超线性的，但是这种情况并不常见。我们在后续章节中给出一个这种应用的例子。

### 12.9.1 Amdahl定律

下面从定量的角度来分析所带来的性能提高。一个计算机系统性能的提高将改善系统的一部分，并不是改善整个系统。提高的性能依赖于改善部分所带来的影响。这个原理被Gene Amdahl用著名的定律<sup>[11]</sup>公式化了。其形式为

654

$$S_{\text{new}} = \frac{\text{Old time}}{\text{New time}} = \frac{1}{1 - f_{\text{enhanced}} + f_{\text{enhanced}} / S_{\text{enhanced}}}$$

式中：

$S_{\text{new}}$  是新系统中的加速，其中包含改善部分；

$S_{\text{enhanced}}$  是系统中改善部分所带来的加速；

$f_{\text{enhanced}}$  是原系统中改善部分所提高的计算时间。

在多处理器系统中，该定律可以按照以下方式重新叙述。 $f$  表示可并行计算的部分（时间形式）， $P$  是系统中处理器的数量， $S_P$  是相对于串行执行时可获得的加速。那么，我们得到：

$$S_P = \frac{1}{1 - f + f/P} = \frac{P}{P - f(P-1)}$$

该公式假定全部处理器使用的都是平衡加载条件下的并行部分。

假定一个给定的应用运行在64处理器的机器上，并且应用中70%是可并行化的，那么可得到的加速是

$$S_{64} = \frac{64}{64 - 0.7 \times 63} = 3.22$$



655

如果同样的应用程序运行在16处理器的机器上,可得到的加速将是2.91。这说明了加速部分远远小于机器中处理器的数量。此外,从16个处理器增加到64个处理器所带来的加速之间的差别是甚微的。显然,对于绝大部分是串行执行(非并行)的应用来说,使用大规模多处理器系统几乎是没有什么意义的。为了得到更高的加速,必须将串行部分变短。对于 $f = 0.95$ 的应用来说,在上述两个机器中可获得的加速分别是15.42和9.14。实际上,Amdahl定律说明的是几乎所有的应用中都有一定的不可并行部分,因此不可能获得线性加速。

这个讨论假设每个处理器都执行相同数量的并行计算。这样相同的平衡加载是不可能发生的。如果在执行下一步操作之前必须等待最慢的处理器完成它的并行任务,那么利用上面的公式计算的结果要比预期的更糟。然而,存在会出现反面结果的应用,也就是只要有一个处理器完成了它的任务,那么所有处理器执行的任务都将终止。例如,这样的异常行为发生在基于一种称为“模拟退火”技术的应用中。为了说明这种技术,假定在VLSI芯片的设计中,希望在芯片上安放逻辑门,这些门的导线总长度将决定电路是否能够达到最小。这就需要尝试大量不同方式的布局方案,以便实现最佳的布局,使全部处理器在同一时刻到达下一迭代的起始点。那么每个处理器就用不同的随机方法来改变逻辑门的位置以寻求最佳布局。只要一个处理器找到了一种比预先起始点都好的位置,该位置就用作所有处理器的新起点,而不必等待其他处理器也去寻找。这类应用展示了超线性的加速,因为在用单处理器处理时,在得到一个较好的位置之前,有大量的时间浪费在尝试那些不可能的情况上。

### 12.9.2 性能指标

从用户的角度来看,一个计算机系统最重要的特点是它的成本、易用性、可靠性和性能。性能指标用来描述计算机的处理能力。8.8节中所讨论的问题也适用于多处理器系统。

处理器的原始能力是通过一秒钟所能进行操作的数量来描述的。两种比较流行的度量标准是MIPS(每秒钟执行指令的百万条数)和MFLOPS(每秒钟执行浮点计算的百万条数)。生产商为特定处理器给出了相应的MIPS和MFLOPS值,这些数字表示的是处理器的最大处理能力。实际中这个最大能力是达不到的。在多处理器系统中,总的MIPS和MFLOPS是系统中所有处理器相应值的总和。

另一个常用性能指标是互连网络的通信能力,通常以每秒字节数的形式给出带宽。这是假定在最理想的情况下,即网络中有足够的传输数据,可以保持网络中的数据传输处于最大的传输状态。这样可以使得每次传送的数据能够达到最大。

虽然MIPS、MFLOPS和网络带宽这些指标对了解系统处理能力是有用的,但是,它们并不能度量应用程序在执行时我们所要观察的性能。实际应用在任何指定的时刻只能使用总资源中的一部分。这部分性能会随着系统与系统的不同以及应用与应用之间的不同而变化。两个不同系统之间的适当比较只能通过一套应用程序在两个系统中同时运行来观察它们的性能。为了满足这种需要,基准程序应运而生。这些程序可以表示普通应用中的各种各样的行为。目前已经普遍采用基准程序进行系统间的性能比较。

### 12.10 结束语

多处理器系统以合理的成本提供了巨型机计算能力的实现途径。它们在几十个处理器到上千个处理器的范围内是最有效的实现方案。由几千个处理器构成的超大规模系统很难充分利用,

同时由于其价格原因而使市场需求大大减少。

实现多计算机系统的一种有效方法是采用局域网将工作站互连起来构成一个系统。这种方法随着局域网速度的提高逐渐变得更具吸引力。

656

多处理器的成功使用很大程度上依赖于能充分使用系统资源的系统软件。如果一个应用程序本身中没有适当使用局部性原理和并行机制,那么它就不会表现出良好的性能。编译器必须能检测出程序中可以并行处理的程序段。同时,操作系统对系统中相邻处理器之间的大量交互,要充分利用局部性原理来调度执行。应用程序员应该对这方面提供有用的线索,不过最好还是由操作系统本身来完成这部分工作。

本章主要对多处理器系统和多计算机系统的最重要特征作了总体上的描述。读者需要学习更多的细节,才能充分理解这些系统的各种功能和所涉及的设计问题。如果您想学习更多的内容,可以参考这方面主题<sup>[12-16]</sup>中的书籍。

## 习题

12.1 编写一个循环程序,可以使图12-1中的控制处理器进行指令的广播,并能使一组处理器重复计算12.2节中讨论的平面的温度。除了相邻处理单元(PE)之间转换网络寄存器内容的指令以外,我们假定有双操作数指令用于PE寄存器和本地内存之间,传送数据并进行算术运算。另外假定每个PE保存着它本地内存CURRENT处的网格点温度和R0、R1等几个处理中使用的寄存器。每个边界PE维持一个固定的网络寄存器中的边界温度值,并且不执行程序广播。在每个PE的EPSILON位置中存储一个很小的数值,是用来判断局部温度是否已经达到所需的精确级别。在循环每次迭代的末尾,每个PE必须设置状态位STATUS。如果新温度满足下面的条件:

$$|\text{New temperature} - [\text{CURRENT}]| < [\text{EPSILON}]$$

STATUS设置成1;否则,STATUS设置成0。

12.2 假定总线传输需要 $T$ 秒,内存访问时间需要 $4T$ 秒。一个经过传统总线的读请求需要 $6T$ 秒完成。在相同的时间延迟下,需要多少根传统的总线才能等于或超过一个分割事务总线的带宽?只考虑读请求,忽略内存冲突并假定所有的内存模块都连接到多总线中的所有总线上。如果内存访问时间增加的话,你的答案是增加还是减少?

12.3 在基于总线的多处理器中,如果系统总线不支持足够高的传输速率,它会成为一个瓶颈。假定一个分割事务总线的宽度设计为系统处理器字长的四倍。如果有效的传输率增加,它的速度是否能增加到带宽与单处理器字长相同的总线的四倍?说明你的理由。

657

12.4 假定在混洗网中 $2 \times 2$ 开关的成本是交叉开关成本的两倍。在 $n \times n$ 交叉开关中有 $n^2$ 个交叉点。随着 $n$ 的增加,交叉开关要比混洗网更加昂贵。当交叉开关成本是混洗网的五倍时, $n$ 最小是多少?

12.5 混洗网可以不用 $2 \times 2$ 的开关,而采用如 $4 \times 4$ 和 $8 \times 8$ 的开关构成。画出一个用 $4 \times 4$ 开关构成的 $16 \times 16$  ( $n = 16$ )的混洗网。如果 $4 \times 4$ 开关的成本是 $2 \times 2$ 成本的四倍,请比较用 $4 \times 4$ 开关与用 $2 \times 2$ 开关构成的混洗网的成本, $n$ 值为数列 $4, 4^2, 4^3, \dots$ ,依次类推。定性比较用两种不同的方式构成的混洗网的阻塞可能性。

12.6 假定PAR段(参见图12-14)中每个程序的执行需要1个时间单位。程序由三个连续段构成。

每个段需要 $k$ 个时间单位并且必须在单处理器上执行。用两个PAR段将三个连续段分开，每个段由 $k$ 个可以在独立的处理器上执行的程序构成。当它在 $n$ 个处理器的多处理器系统上运行时，给出该程序的加速表达式，假定 $n < k$ 。当 $k$ 增大且 $n = k$ 时，加速的极限值是多少？从这个结果中，你知道实际上带有并行能力的程序中连续段的作用是什么吗？

- 12.7 在一个 $n$ 维超立方体中消息传输的最短距离是1步跳跃，最长距离是 $n$ 步跳跃。假定所有的源/目的对是相等的，消息传输的距离是大于 $(1+n)/2$ 还是小于 $(1+n)/2$ ？证明你的结论。
- 12.8 在一个双指令循环（如图12-15所示）中，利用测试-置位指令对一个锁变量进行“忙等待”操作时浪费了本可以用于计算的总线周期。针对该问题，请给出一种由操作系统维护的“集中式等待任务队列”的解决方案。假定操作系统可以由用户任务调用，并且操作系统可以从等待执行的任务中进行选择并分配给一个处理器执行。
- 12.9 在高速缓存一致性中支持和反对无效化策略与修改策略的观点分别是什么？
- 12.10 12.6.3节中讨论的高速缓存一致性控制并不能代替锁变量。那么，利用锁变量能够代替高速缓存一致性控制吗？
- 658 12.11 如果不采用图12-18中的程序而是采用图12-19中的程序，估算一下性能的改善情况。假定执行程序的每一步所需的时间一定。
- 12.12 修改图12-19中的程序，使之适合于在4处理器机器上执行。
- 12.13 修改图12-20中的程序，使之适合于在4处理器机器上执行。
- 12.14 对于小型向量，用图12-19中的方法计算点积不如使用单一处理器。估算一下该方法具有更高性能时向量的最小规模。假定执行程序的每一步所需的时间一定。
- 12.15 用图12-20中的方法重复12.14中的问题。
- 12.16 共享存储器多处理器和消息传递多计算机都是支持同时执行交互任务的体系结构。这两种体系结构中哪种更容易模拟另一种的动作？简要说明你的理由。
- 12.17 以太网总线局域网协议在消息传送时间远远大于 $2\tau$ （其中 $\tau$ 是数据从总线的一端到另一端的传输延迟）时是最适合的。可能存在这种情况吗？即使源节点在 $2\tau$ 的冲突窗口内检测到冲突时，目的节点仍能正确接收到一个非失真的消息吗？如果不能，说明你的理由。如果你认为可能，请给出总线上源节点、目的节点和干扰节点之间的相对位置，并描述相关事件的时间。
- 12.18 邮箱式存储器是一个具备如下特征的RAM：满/空位（F/E）与每个内存字位置相关。指令

PUT R0, BOXLOC, WAITSEND

按照如下方式执行。与邮箱式存储器位置BOXLOC相关的F/E位用来测试。如果F/E为0表示空，那么寄存器R0中的内容将被写到BOXLOC中；如果F/E为1表示满，就执行下面的串行指令。否则（也就是F/E=1）不执行任何操作，并且将执行控制传递给在程序内存中且位置为WAITSEND的指令。

(a) 请给指令

GET R0, BOXLOC, WAITREC

659

下一个适当的定义，作为对PUT指令的补充。

(b) 假定在一个多处理器系统中不同处理器上运行的两个任务 $T_1$ 和 $T_2$ ，利用PUT和GET指令在共享的邮箱式存储器上由 $T_1$ 向 $T_2$ 传送一个单字消息的流。用汇编语言为 $T_1$ 和 $T_2$ 编写

程序段, 在没有邮箱式存储器但是具有TAS指令的共享存储器处理器系统中实现相同的功能。

## 参考文献

1. M.J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54, December 1966, pp. 1901-1909.
2. D.L. Slotnick, "The Fastest Computer," *Scientific American*, vol. 224, February 1971, pp. 76-88.
3. D. Lenoski, et al., "The Stanford DASH Multiprocessor," *Computer*, vol. 25, March 1992, pp. 63-79.
4. J. Kuskin, et al., "The Stanford FLASH Multiprocessor," *Proceedings of the 21st Annual International Symposium on Computer Architecture*, Chicago, April 1994, pp. 302-313.
5. A. Agarwal, et al., "The MIT Alewife Machine: Architecture and Performance," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 2-13.
6. Z.G. Vranesic, M. Stumm, D.M. Lewis, and R. White, "Hector: A Hierarchically Structured Shared-Memory Multiprocessor," *Computer*, vol. 24, January 1991, pp. 72-79.
7. R. Grindley, et al., "The NUMachine Multiprocessor," *Proceedings of the 2000 International Conference on Parallel Processing*, Toronto, Ont., August 2000, pp. 487-496.
8. W.J. Dally and P. Song, "Design of a Self-Timed Multicomputer Communication Controller," *Proceedings of the 1987 International Conference on Computer Design*, October 1987, pp. 230-234.
9. D. Gustavson, "The Scalable Coherent Interface and Related Standards Projects," *IEEE Micro*, vol. 12, January 1992, pp. 10-22.
10. *IEEE Local Area Standard 802*, IEEE, 1985.
11. G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities," *Proceedings of AFIPS Spring Joint Computer Conference*, Atlantic City, NJ, April 1967, pp. 483-485.
12. D. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture — A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, CA, 1998.
13. G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, 2nd ed., Benjamin-Cummings, Redwood City, CA, 1994.
14. K. Hwang, *Advanced Computer Architecture*, McGraw-Hill, New York, 1993.
15. H.S. Stone, *High-Performance Computer Architecture*, 3rd ed., Addison-Wesley, Reading, MA, 1993.
16. D. Tabak, *Multiprocessors*, Prentice-Hall, Englewood Cliffs, NJ, 1990.



## 逻辑电路

数字计算机中的信息是由被称为逻辑电路的电子网络表示和处理的。这些电路对二进制变量进行操作，我们可以为二进制变量赋以两个不同的值，通常是0和1。本附录将对逻辑函数和实现电路做简要描述，并对集成电路技术做简单介绍。

## A.1 基本逻辑函数

下面用一个所有家庭都会发生的实际问题来介绍二进制逻辑。来看一个由两个开关 $x_1$ 、 $x_2$ 控制开/关状态的灯泡。每个开关都可以处于两个可能的位置：0或1中的任意一个，如图A-1a所示。这样它就可以用一个二进制变量表示。我们将开关的名称作为其对应二进制变量名称。图中还有一个电源和一个灯泡。开关端口的连接方式决定了开关控制灯的方式。只有存在一个从电源经开关网络到灯泡的闭合回路时，灯才会亮。我们用二进制变量 $f$ 表示灯的状态。如果灯亮， $f=1$ ；如果灯不亮， $f=0$ 。这样， $f=1$ 说明电路中至少有一个闭合回路；而 $f=0$ 说明没有闭合回路。显然， $f$ 是变量 $x_1$ 和 $x_2$ 的函数。

661  
662

下面看一些控制灯状态的可能情况。首先，假设任一开关在1的位置时灯都会亮，即当

$$x_1 = 1 \text{ 且 } x_2 = 0$$

或

$$x_1 = 0 \text{ 且 } x_2 = 1$$

或

$$x_1 = 1 \text{ 且 } x_2 = 1$$

时， $f=1$ 。

实现这种控制的电路连接方式如图A-1b所示。电路图旁的逻辑真值表表示了这一情况。表中列出了所有可能的开关状态及在每个状态下的 $f$ 值。在逻辑术语中，这个表格表示了两个变量 $x_1$ 和 $x_2$ 的“或”（OR）函数。这个操作在代数中用符号“+”或符号“ $\vee$ ”表示，即

$$f = x_1 + x_2 = x_1 \vee x_2$$

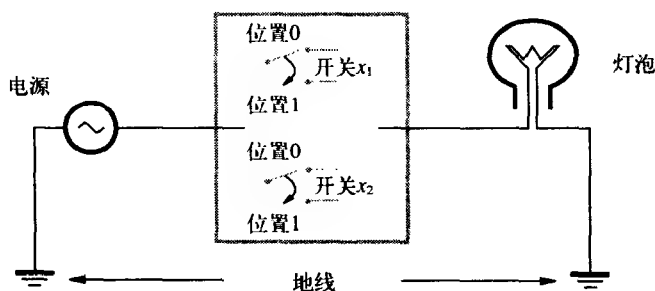
我们称 $x_1$ 、 $x_2$ 为输入变量， $f$ 为输出函数。

下面是OR操作的一些基本特性。它是可交换的，即

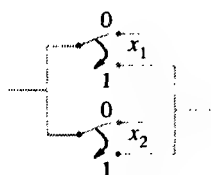
$$x_1 + x_2 = x_2 + x_1$$

这一性质可扩展到 $n$ 变量中，即

$$f = x_1 + x_2 + \cdots + x_n$$

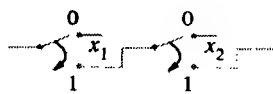


a) 由两个开关控制的灯泡



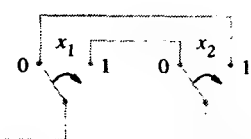
$x_1$	$x_2$	$f(x_1, x_2) = x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

b) 并联 (“或” 控制)



$x_1$	$x_2$	$f(x_1, x_2) = x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

c) 串联 (“与” 控制)



$x_1$	$x_2$	$f(x_1, x_2) = x_1 \oplus x_2$
0	0	0
0	1	1
1	0	1
1	1	0

d) 异或联接 (“异或” 控制)

图A-1 电灯开关示例

如果任意一个 $x_i$ 的值为1,  $f$ 的值就为1。这反映了在图A-1b中的两个开关上并联更多开关时的效果。而且, 通过观察真值表可以看出

$$1 + x = 1$$

且

$$0 + x = x$$

现在, 假设只有当两个开关都在1的位置时灯才会亮。这个状态的电路连接和相应的真值表表示如图A-1c。这种情况为“与”(AND)操作, 使用符号“ $\cdot$ ”或“ $\wedge$ ”表示。即

$$f = x_1 \cdot x_2 = x_1 \wedge x_2$$

与操作的一些基本性质有

$$x_1 \cdot x_2 = x_2 \cdot x_1$$

$$1 \cdot x = x$$

及

$$0 \cdot x = 0$$

与函数也可扩展到 $n$ 个变量

$$f = x_1 \cdot x_2 \cdots x_n$$

上式只有当所有的 $x_i$ 都取1时才得1。这反映了在图A-1c中两个开关上串联更多的开关而产生的效果。

要讨论的最后一种控制灯状态的开关方式是另一种常见的情况。假设走廊的两端都有开关，每个开关都应该能控制灯的开或关。即：如果灯是亮的，改变任一个开关的位置都应将其关闭；而如果灯是灭的，改变任一个开关的位置都应将其打开。假设两个开关都在0位置时，灯是灭的。那么将任一个开关置为1都能将灯打开。现在假定灯是亮的，且 $x_1 = 1, x_2 = 0$ 。显然将开关 $x_1$ 变为0可以把灯关闭。而且，将 $x_2$ 置为1也能将灯关闭，即当 $x_1 = x_2 = 1$ 时， $f = 0$ 。实现这种控制的电路如图A-1d所示。相应的逻辑操作称为“异或”（EXCLUSIVE-OR, XOR）操作，使用符号“ $\oplus$ ”表示。它的性质有

$$x_1 \oplus x_2 = x_2 \oplus x_1$$

$$1 \oplus x = \bar{x}$$

及

$$0 \oplus x = x$$

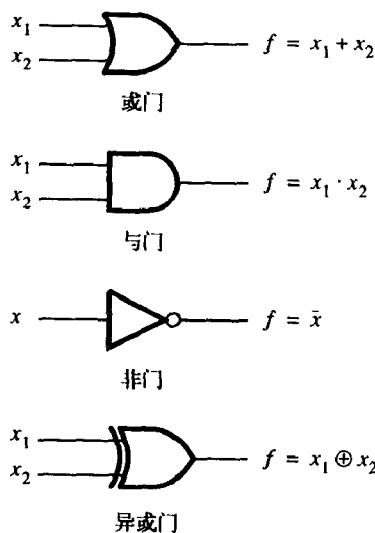
其中 $\bar{x}$ 代表 $x$ 的“非”操作（NOT）。单变量的函数 $f = \bar{x}$ ，当 $x = 0$ 时取1，当 $x = 1$ 时取0。我们称输入 $x$ 取反或求补了。

## 电子逻辑门

因为具有相似性和简单性，使用开关、闭合或断开电路、以及用灯泡来说明逻辑变量和函数的思想是非常方便的。前面介绍的逻辑概念同样适用于数字计算机中处理信息的电子电路。但物理变量是电路的电平和电流，而不是开关的位置以及闭合或断开的电路。例如，考虑一个电路，其输入电平为+5伏或0伏。电路输出电平同样是+5伏或0伏。现在，如果用+5伏代表逻辑1，0伏代表逻辑0，那么就可以指定该电路逻辑操作的真值表，从而描述电路的作用。

借助于晶体管，我们可以设计出简单的电路完成与、或、异或以及非操作。习惯上将这些基本逻辑电路称为门。图A-2给出了这些门的标准符号。在逻辑门的输入或输出反相时，我们使用一种更为紧凑的图形记号来表示非操作。在这种情况下，反相用一个小圆圈表示。

逻辑门的电路实现将在A.5节中进行讨论。接下来继续讨论怎样使用逻辑门构造一些具有更复杂逻辑功能的逻辑网络。



图A-2 标准逻辑门符号

664

665

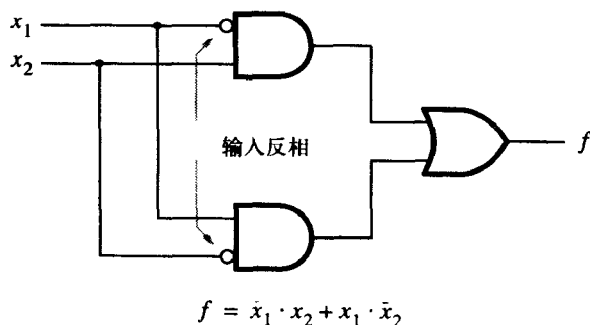


## A.2 逻辑函数的组合

考虑图A-3a中由两个与门和一个或门组成的网络。它可以用下面的式子表示

$$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

图A-3b是这个表达式的真值表。首先，与门的值由每个输入值决定。而函数 $f$ 的值由或操作决定。 $f$ 的真值表与异或函数的真值表一样，所以图A-3a使用与门、或门、非门实现了异或函数。逻辑表示式 $\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$ 称为积之和的形式，因为OR操作有时被称作“和”函数，而AND操作被称作“积”函数。



$$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$$

a) 异或函数网络

$x_1$	$x_2$	$\bar{x}_1 \cdot x_2$	$x_1 \cdot \bar{x}_2$	$f = \bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$ $= x_1 \oplus x_2$
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

b)  $\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2$  的真值表

图A-3 使用与门、或门、非门实现异或函数

需要注意的是，为了说明执行操作的顺序，将表达式写为 $f = ((\bar{x}_1) \cdot x_2) + (x_1 \cdot (\bar{x}_2))$ 更为合适。为了简化表达式，我们需要对与、或、非三种操作的优先级进行定义。在没有圆括号时，逻辑表达式的操作应当按照以下顺序执行：非、与，然后是或。此外，在不产生歧义的情况下习惯上省略“ $\cdot$ ”运算符。

回到积之和的形式，我们现在解释如何直接从真值表推导出任意逻辑函数的积之和形式。考虑表A-1中的真值表，并假设用与门、或门、非门组成函数 $f_1$ 。对于表中 $f_1 = 1$ 的每一行，都引入积之和形式中的一个乘积（与）项。这个乘积项包括所有三个输入变量。非操作作用在单个变量上，使得只有当这些变量取与真值表的那一行对应的某个确定值时，乘积项才为1。这表明如果 $x_i = 0$ ，则 $\bar{x}_i$ 就在乘积项中；如果 $x_i = 1$ ，则 $x_i$ 在乘积项中。例如，表中第4行的函数值为1，而输入变量为

$$(x_1, x_2, x_3) = (0, 1, 1)$$

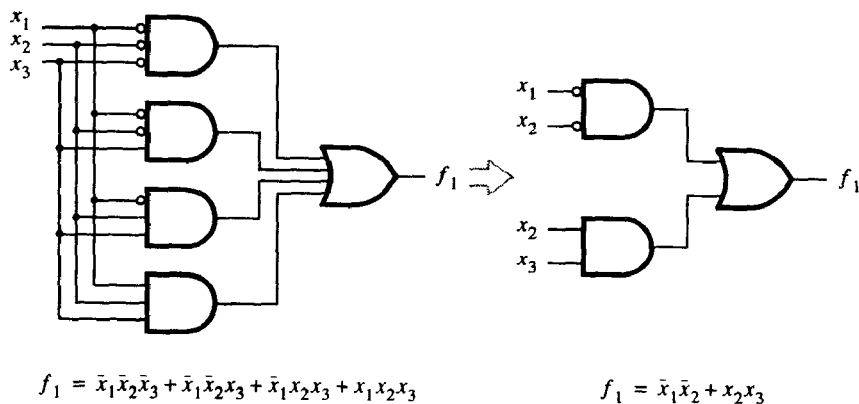
则相应的乘积项是  $\bar{x}_1 x_2 x_3$ 。将所有  $f_1$  为1的行都进行同样的操作, 得到

$$f_1 = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3$$

表A-1 两个3变量函数

$x_1$	$x_2$	$x_3$	$f_1$	$f_2$
0	0	0	1	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

与这一表达式相对应的逻辑网络在图A-4的左方。另外一个例子, 异或函数的积之和表达式可以使用这个方法由其真值表导出。这个方法可以从任意大小的真值表导出积之和表达式以及相应的逻辑网络。



图A-4 表A-1中  $f_1$  的逻辑网络及与其等价的最小网络

### A.3 逻辑表达式的化简

前面介绍了如何从真值表导出一个积之和表达式。实际上, 对于任意一个特定的真值表都有许多等价的表达式和逻辑网络。两个逻辑表达式或逻辑门电路如果具有相同的真值表, 那么它们就是等价的。与上一节导出的  $f_1$  的积之和表达式等价的一个表达式是

$$\bar{x}_1 \bar{x}_2 + x_2 x_3$$

要证明这一点, 我们在表A-2中列出这个简单表达式的真值表, 并表明它与表A-1中  $f_1$  函数的真值表是相同的。创建  $\bar{x}_1 \bar{x}_2 + x_2 x_3$  真值表的过程分为三步。首先, 计算每种输入值下乘积项  $\bar{x}_1 \bar{x}_2$  的值, 然后是乘积项  $x_2 x_3$ 。最后, 将这两列相“或”得到表达式的真值表。这个真值表与表A-1的  $f_1$  函数真值表完全一致。

为了简化逻辑表达式，我们需要进行一系列的代数操作。这些操作所使用的新的逻辑规则是分配律

$$w(y+z) = wy + wz$$

和恒等式

$$w + \bar{w} = 1$$

表A-2 表达式 $\bar{x}_1\bar{x}_2 + x_2x_3$ 的计算

$x_1$	$x_2$	$x_3$	$\bar{x}_1\bar{x}_2$	$x_2x_3$	$\bar{x}_1\bar{x}_2 + x_2x_3 = f_1$
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

表A-3给出了分配律的真值表证明。现在应该清楚，像这样的规则总是可以通过列出其左边和右边的真值表，然后显示它们相同的方式来证明。逻辑规则如分配律，有时被称为恒等式。尽管在这里用不到，但为了完整性，再给出分配律的另一种形式：

$$w + yz = (w + y)(w + z)$$

表A-3 真值表方法证明表达式等价

$w$	$y$	$z$	$y+z$	等号左边 $w(y+z)$	$wy$	$wz$	等号右边 $wy + wz$
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

逻辑化简的目的是根据一些标准减少实现特定逻辑函数的成本。特别地，我们希望从由真值表导出的积之和表达式开始，将它化简为最小积之和表达式。为了定义化简的标准，有必要引入一个衡量积之和表达式的大小或成本的量度。通常成本量度是以图A-4的形式实现表达式所需的门和门输入的总数。例如，图中较大的表达式成本为21，由5个门和16个门输入组成。在计数过程中忽略了输入的反相。较小的表达式成本为9，由3个门和6个门输入组成。现在可以明确，如果没有其他等价的积之和表达式具有更小的成本，那么这个积之和表达式就是最小的。下面所举的简单例子可以清楚地看出我们得到了最小的表达式。因此不再给出最小化的严格证明。

简化一个给定表达式的一般算术操作如下。首先，将只有一些变量在一个乘积项中为反( $\bar{x}$ )，而在另一个乘积项中为真( $x$ )，并且其他变量相同的项结合为一组。根据分配率，提出由其他变量组成的公共乘积子项后，则剩下 $x + \bar{x}$ 项，其值为1。对第一个 $f_1$ 的表达式应用这一操作，我们得到：

$$\begin{aligned}
 f_1 &= \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 x_3 + x_1 x_2 x_3 \\
 &= \bar{x}_1 \bar{x}_2 (\bar{x}_3 + x_3) + (\bar{x}_1 + x_1) x_2 x_3 \\
 &= \bar{x}_1 \bar{x}_2 \cdot 1 + 1 \cdot x_2 x_3 \\
 &= \bar{x}_1 \bar{x}_2 + x_2 x_3
 \end{aligned}$$

这个表达式是最小的。相应的网络连接如图A-4所示。

表A-4 二进制逻辑规则

名 称	算 术 等 式	
交换律	$w + y = y + w$	$wy = yw$
结合律	$(w + y) + z = w + (y + z)$	$(wy)z = w(yz)$
分配律	$w + yz = (w + y)(w + z)$	$w(y + z) = wy + wz$
幂等律	$w + w = w$	$ww = w$
对合律	$\overline{\overline{w}} = w$	
互补律	$w + \bar{w} = 1$	$w\bar{w} = 0$
德摩根律	$\overline{w + y} = \bar{w}\bar{y}$	$\overline{wy} = \bar{w} + \bar{y}$
	$1 + w = 1$	$0 \cdot w = 0$
	$0 + w = w$	$1 \cdot w = w$

670

将乘积项结对，从而便得到最简表达式的最小化过程并不总是像前面例子那样明显。一个很有用的规则是

$$w + w = w$$

这个规则允许我们重复使用乘积项，从而在提取公因式的过程中，某个乘积项可与其他多个项组合。举个例子，看一下表A-1中的 $f_2$ 函数。其积之和表达式可直接由真值表导出为

$$f_2 = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{x}_2 x_3$$

通过重写第一个乘积项 $\bar{x}_1 \bar{x}_2 \bar{x}_3$ 并交换项的位置（根据交换律），我们得到

$$f_2 = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 \bar{x}_2 x_3 + \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3$$

将乘积项结对并提取公因子

$$\begin{aligned}
 f_2 &= \bar{x}_1 \bar{x}_2 (\bar{x}_3 + x_3) + x_1 \bar{x}_2 (\bar{x}_3 + x_3) + \bar{x}_1 (\bar{x}_2 + x_2) \bar{x}_3 \\
 &= \bar{x}_1 \bar{x}_2 + x_1 \bar{x}_2 + \bar{x}_1 \bar{x}_3
 \end{aligned}$$

前两项再次通过因式分解化去，然后得到最小表达式

$$f_2 = \bar{x}_2 + \bar{x}_1 \bar{x}_3$$

这样就完成了我们对逻辑表达式的算术化简的讨论。这项数学练习有一个明显的实际应用，就是由较少的门和输入构成的网络更加廉价并且易于实现。因此，确定与所给表达式等价的最简表达式符合经济利益的需要。表A-4中总结了操作逻辑表达式时应用的规则。它们成对出现，显示出对称性，因为它们对“与”和“或”函数同样适用。到目前为止，我们还没有机会使用对合律或德摩根律，但是在下一节中会发现它们很有用。

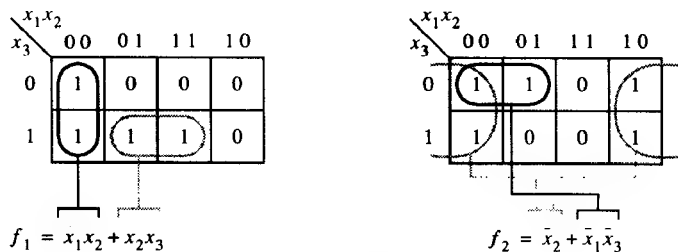
### A.3.1 使用卡诺图化简

在对表A-1中的函数 $f_1$ 和 $f_2$ 进行代数化简的过程中，在某一时刻必须猜测最佳的处理方法。

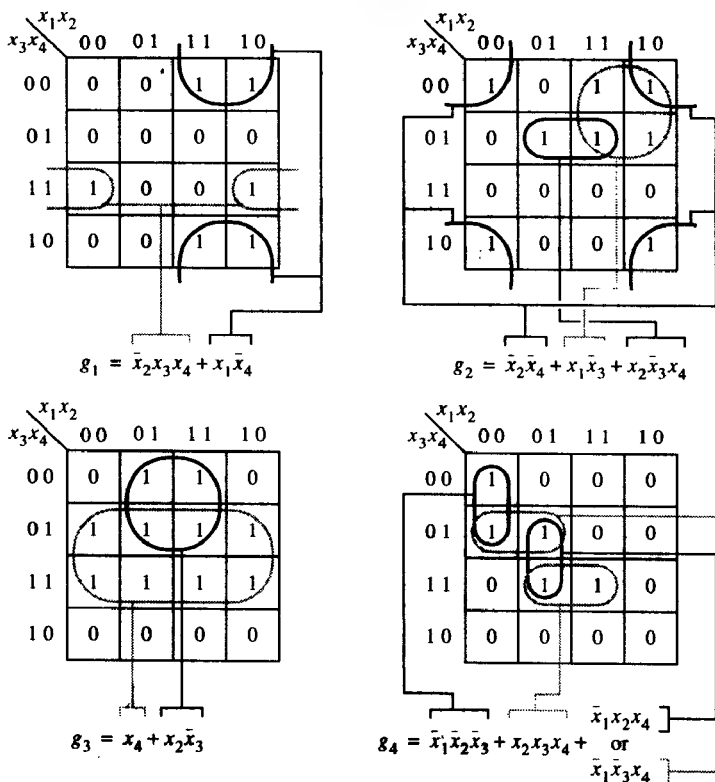
例如, 在  $f_2$  的化简中判断第一步重写项  $\bar{x}_1 \bar{x}_2 \bar{x}_3$  并不明显。有一种几何方法可以迅速地导出含有几个变量的逻辑函数的最小表达式。这一技术依赖于真值表的另一种表示形式, 称为卡诺图。对于一个3变量函数, 它是一个由2行4列共8个方块组成的长方形, 如图A-5a所示。每个方块对应于输入变量的一组特定的值。例如, 第一行的第三个方块表示值  $(x_1, x_2, x_3) = (1, 1, 0)$ 。因为3变量真值表为8行, 所以卡诺图显然需要8个方块, 方块中的内容是与输入变量值相对应的函数值。

构建卡诺图的关键思想是水平和垂直相邻的方块对应着只有一个变量不同的输入变量值。当两个相邻的方块值都为1, 表明可以进行代数化简。

671



a) 3变量图



b) 4变量图

图A-5 使用卡诺图进行化简

在图A-5a函数  $f_2$  的卡诺图中, 第一行最左边的两个值为1的方块与乘积项  $\bar{x}_1 \bar{x}_2 \bar{x}_3$ 、 $\bar{x}_1 x_2 \bar{x}_3$  对应。在最小化  $f_2$  代数表达式过程中先执行化简

$$\bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 = \bar{x}_1 \bar{x}_3$$

672

将图中的两个1格圈为一组就可以直接得到化简结果。与一组方块相对应的乘积项是在这些方块中值不变的那些输入变量的乘积。如果 $x_i$ 在一组1格中的值为0, 则将 $\bar{x}_i$ 加入乘积项, 但如果 $x_i$ 的值是1, 则将 $x_i$ 加入乘积项。两方块相邻的情况包括最左端的方块与最右端的方块相邻。继续我们对 $f_2$ 的讨论, 由最左列和最右列组成的4个1格的组化简得到单一变量项 $\bar{x}_2$ , 因为 $x_2$ 是该组中惟一一个值不变的变量。其他两个变量的所有4种可能的组合都在该组中出现过了。

卡诺图可以用于多于3个变量的情况。4变量的卡诺图可以由两个3变量的卡诺图得到。图A-5b是4变量卡诺图的例子, 同时还给出了图中表示的函数的最小表达式。除了2个方块和4个方块的组外, 现在还可以组建8个方块的组。这样的分组在 $g_3$ 的图中有所显示。注意 $g_2$ 中角上的4个方块也构成了一个合法的4格组, 代表乘积项 $\bar{x}_2\bar{x}_4$ 。与3变量的图一样, 与一组方块所对应项是组中没有改变值的变量乘积。例如 $g_2$ 图中右上角4个1格的组表示为乘积项 $x_1\bar{x}_3$ , 因为组中 $x_1 = 1$ ,  $x_3 = 0$ 。该组中包含了变量 $x_2$ 与 $x_4$ 所有可能的取值组合。对于5变量函数也可以使用卡诺图。在这种情况下, 将使用两个4变量卡诺图, 其中一个对应于第5个变量取0的情况, 另一个对应取1的情况。

673

可以很容易地得出在卡诺图中划分2格组、4格组、8格组等的一般方法。两个相邻的值为1的对可以组合为一个4格组。类似地, 两个相邻的4格组可以组成一个8格组。通常, 任意有效组中方格的个数一定为 $2^k$ , 其中 $k$ 为整数。

我们现在考虑使用卡诺图得到最小积之和表达式的过程。从图A-5中可以看出, 大的值为1的组对应于小的乘积项。这样, 要得到简单门电路, 图中所有1的方块分组越少越好。一般说来, 我们应当选择组最少的分法, 选择那些尽可能大的包含图中所有1的组。例如, 考虑图A-5b中的函数 $g_2$ 。如图所示, 四个角上值为1的方块组成了表示乘积项 $\bar{x}_2\bar{x}_4$ 的组。另一个4格组在右上角, 表示乘积项 $x_1\bar{x}_3$ 。这两个组包括了除位置为 $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$ 外所有值为1的方块。包含这个方块的最大的组是一个2格组, 表示乘积项 $x_2\bar{x}_3x_4$ 。因此,  $g_2$ 的最小表达式是

$$g_2 = \bar{x}_2\bar{x}_4 + x_1\bar{x}_3 + x_2\bar{x}_3x_4$$

图中其他函数的最小表达式可使用类似方法得到。注意 $g_4$ 有两种可能的最小表达式, 一个包括 $\bar{x}_1x_2x_4$ 项, 另一个包括 $\bar{x}_1\bar{x}_3x_4$ 项。经常会出现一个给定的函数有不只一个最小表达式的情况。

在我们举的所有例子中, 得到最小表达式都比较容易。通常这一过程有正式的算法, 但这里不再讨论。

### A.3.2 无关项条件

在许多情况下, 数字电路的一些输入值永远都不会发生。例如, 考虑二进制编码的十进制数(BCD)表示法。四个二进制变量 $b_3$ 、 $b_2$ 、 $b_1$ 和 $b_0$ 表示十进制数0到9, 如图A-6所示。这4个变量总共有16个不同的值, 其中只有10个用于表示十进制数。剩下的值没有用到。因此, 任何处理BCD码的逻辑电路都永远不会在其输入中碰到这6个值中的任何一个。

图A-6给出了一个操作BCD数的特定函数的真值表。我们并不关心那些从不使用的输入值的函数值是多少; 因此, 它们被称为无关项, 在真值表中用字母“d”表示。为了实现一个电路, 对应于约束项的函数值可被任意赋值为0或1。最佳的赋值方法将得到最小的逻辑门实现。当无关项能够扩大一个1值的组时, 我们就把它赋值为1。因为较大的组对应着较小的乘积项, 适当地使用无关项能够加强最小化的程度。

图A-6中的函数表示了如下对输入十进制数的处理过程: 当输入是一个可被3整除的非零数时, 输出 $f$ 为1。我们需要3个组来包含图中的3个值为1的方块, 并且尽可能使用无关项来扩大组。

#### A.4 与非门、或非门的组合

现在来看另外两个被称为与非门 (NAND) 和或非门 (NOR) 的基本逻辑门, 由于它们的电路实现很简单, 因此在实践中被广泛使用。这些门的真值表如图A-7所示。它们等价地实现了“非”函数下的“与”和“或”函数, 这便是它们的名称以及标准逻辑符号的由来。我们用键头“↑”和“↓”代表与非及或非运算符, 并使用表A-4中的德摩根律, 可以得到

$$x_1 \uparrow x_2 = \overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$$

和

$$x_1 \downarrow x_2 = \overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$$

存在多于两个输入的与非门和或非门, 根据德摩根律的简单扩展可以得到

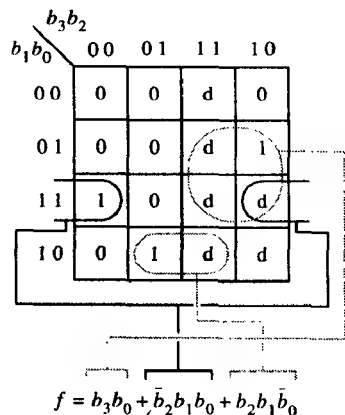
$$x_1 \uparrow x_2 \uparrow \cdots \uparrow x_n = \overline{x_1 x_2 \cdots x_n} = \bar{x}_1 + \bar{x}_2 + \cdots + \bar{x}_n$$

和

$$x_1 \downarrow x_2 \downarrow \cdots \downarrow x_n = \overline{x_1 + x_2 + \cdots + x_n} = \bar{x}_1 \bar{x}_2 \cdots \bar{x}_n$$

使用与非门和或非门进行逻辑设计不像使用与、或、非门那样直截了当。设计过程的主要难点之一是结合律对与非和或非操作是无效的。我们稍后再对这一问题展开讨论。先来讨论只使用与非门实现任意逻辑函数的简单过程。将一个积之和形式的逻辑网络变换成相应

十进制数表示	二进制编码 $b_3 b_2 b_1 b_0$	$f$
0	0 0 0 0	0
1	0 0 0 1	0
2	0 0 1 0	0
3	0 0 1 1	1
4	0 1 0 0	0
5	0 1 0 1	0
6	0 1 1 0	1
7	0 1 1 1	0
8	1 0 0 0	0
9	1 0 0 1	1
未使用	1 0 1 0	d
	1 0 1 1	d
	1 1 0 0	d
	1 1 0 1	d
	1 1 1 0	d
	1 1 1 1	d



$$f = b_3 b_0 + \bar{b}_2 b_1 b_0 + b_2 b_1 \bar{b}_0$$

图A-6 带无关项的4变量卡诺图

$x_1$	$x_2$	$f$
0	0	1
0	1	1
1	0	1
1	1	0

$x_1$	$x_2$	$f$
0	0	1
0	1	0
1	0	0
1	1	0

$$f = x_1 \uparrow x_2 = \overline{x_1 x_2} = \bar{x}_1 + \bar{x}_2$$

$$f = x_1 \downarrow x_2 = \overline{x_1 + x_2} = \bar{x}_1 \bar{x}_2$$



a) 与非门



b) 或非门

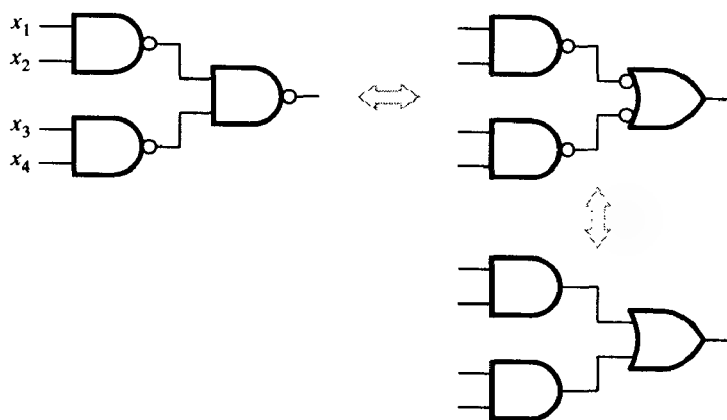
图A-7 与非门和或非门

只由与非门组成的网络有一个直接的方法。借助一个例子可以很容易地了解这个过程。考虑下面对应着由三个2输入与非门组成的4输入网络的逻辑表达式的代数运算过程。

$$\begin{aligned}(x_1 \uparrow x_2) \uparrow (x_3 \uparrow x_4) &= \overline{\overline{(x_1 x_2)} \overline{(x_3 x_4)}} \\ &= \overline{\overline{x_1 x_2} + \overline{x_3 x_4}} \\ &= x_1 x_2 + x_3 x_4\end{aligned}$$

在推导过程中我们已经使用了德摩根律和对合律。图A-8给出了与这个推导对应的逻辑网络。因为任何逻辑函数都可以表示为“积之和”(与-或)形式,而且前述的导出过程完全可逆。我们得出结论,任何逻辑函数都可以表示为“与非-与非”(NAND-NAND)形式。可以看到这一结论对含有任意数量变量的函数都是正确的。与非门的输入数显然与相应的与门和或门的输入数相同。

676



图A-8 等价的与非-与非和与-或网络

让我们回到关于结合律对与非操作符不适用这一话题上来。在图A-8的步骤中使用与非门对逻辑网络进行设计时,与非门所需的输入数目可能会比标准商用与非门输入数多。如果使用与门和或门实现,这就不成问题,因为与和或操作符是可结合的,可以将有限扇入的门直接级联起来。使用2输入门实现3输入与函数和或函数的情况如图A-9a所示。使用与非门的解决方案就不这么简单了。例如,一个3输入与非函数不能用两个2输入与非门级联实现。而需要三个门,如图A-9b所示。

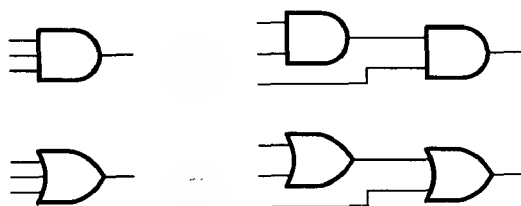
对只使用或非门实现逻辑函数的讨论与前面类似。任意逻辑函数都可以表示为“和之积”(或-与)的形式。这样的网络可以由等价的或非-或非网络实现。

前面的讨论介绍了一些逻辑设计的基本概念。这一主题的详细讨论可在许多教科书中找到(参见参考文献[1、3、7~11])。

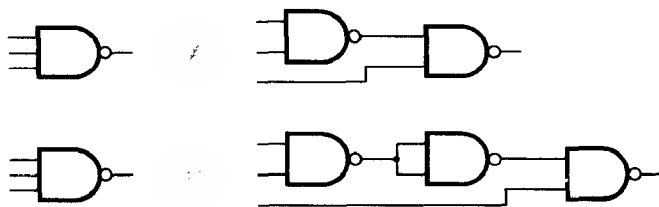
677

需要明确的是,一个给定的逻辑函数可能有许多不同的实现。由于实际应用的需要,我们要找到成本最小的实现方法,也经常需要在逻辑网络中减小传输延迟。为了描述逻辑组合的性质和尽可能地降低成本,前几节中介绍了最小化的概念。例如,卡诺图图形化地表明了得到最佳结果的操作可能性。尽管理解逻辑网络的最优化法则很重要,但并不需要人工实现最优化。复杂的计算机辅助设计(CAD)程序可以完成这些集成。设计者只需要指定所需的功能行为,CAD软件就会给出一个实现这一功能的经济且高效的网络。





a) 用2输入门实现3输入与函数和或函数



b) 用2输入门实现3输入与非函数

图A-9 门的级联

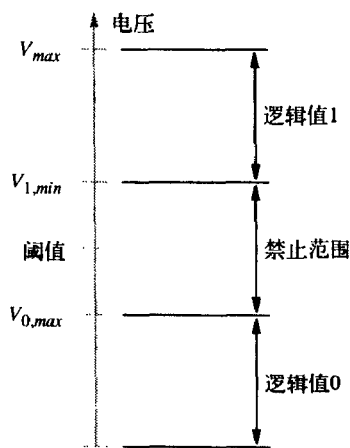
## A.5 逻辑门的实现

现在让我们关注在实践中表示逻辑变量和实现逻辑函数的方法。表示逻辑变量的物理参数的选择明显依赖于具体的技术。在电路里，电平或电流值都可以用于此目的。

为了建立电平和逻辑值或状态的对应关系，我们引入了阈值这一概念。高于给定阈值的电平表示一个逻辑值，而低于阈值的电平表示另一个。在实际情况下，由于各种各样的原因，电路中任何一点的电平都有小的随机变化。因为存在“噪声”，我们不能确定阈值附近电平的逻辑状态。如图A-10所示，为了避免出现这样的不确定性，应该确定一个“禁止范围”。这样，低于 $V_{0,max}$ 的电平表示0值，高于 $V_{1,min}$ 的电平表示1值。在下面的讨论中，将经常用到“低”和“高”这两个术语来分别表示与逻辑值0和1对应的电平。

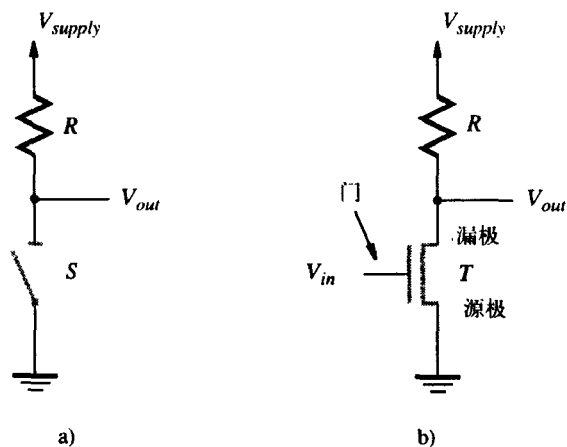
**678** 我们对实现基本逻辑函数的电子线路的讨论将从简单的由电阻和作为开关使用的晶体管组成的电路开始。考虑图A-11中的电路。当图A-11a中的开关 $S$ 关闭时，输出电平 $V_{out}$ 等于0（接地）。当 $S$ 打开时，输出电平 $V_{out}$ 等于提供的电平 $V_{supply}$ 。图A-11b用晶体管 $T$ 代替了开关 $S$ ，也可以得到同样的效果。当提供给晶体管的门输入电平为0时（即 $V_{in} = 0$ ），晶体管相当于一个打开的开关，且 $V_{out} = V_{supply}$ 。当 $V_{in}$ 变化为 $V_{supply}$ 后，晶体管相当于一个闭合的开关，输出电平 $V_{out}$ 非常接近0。因此，电路实现了一个逻辑非门的功能。

现在可以讨论更为复杂的逻辑函数的实现。图A-12显示了或非门的实现电路。在这种情况下，只有当开关 $S_a$ 和 $S_b$ 都打开时，图A-12中的 $V_{out}$ 才是高电平。与之类似，图A-12b中只有输入电平 $V_a$ 和 $V_b$ 都是低电平时， $V_{out}$ 才为高电平。因此，该电路对应一个或非门，其中 $V_a$ 和 $V_b$ 对应于两个输入变量 $x_1$ 、 $x_2$ 。

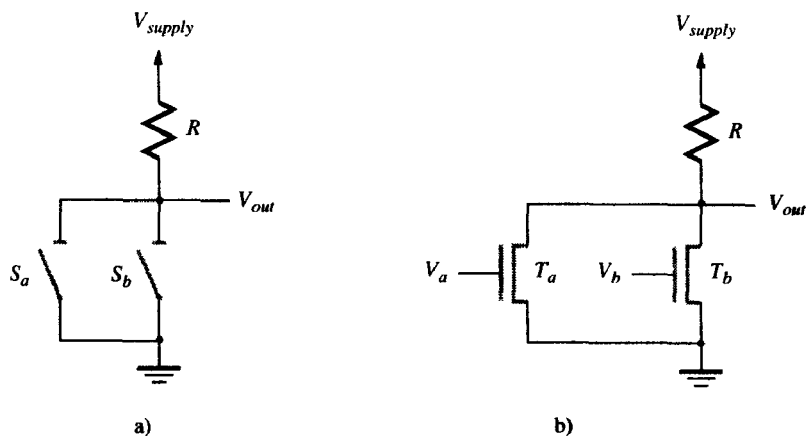


图A-10 用电平表示逻辑值

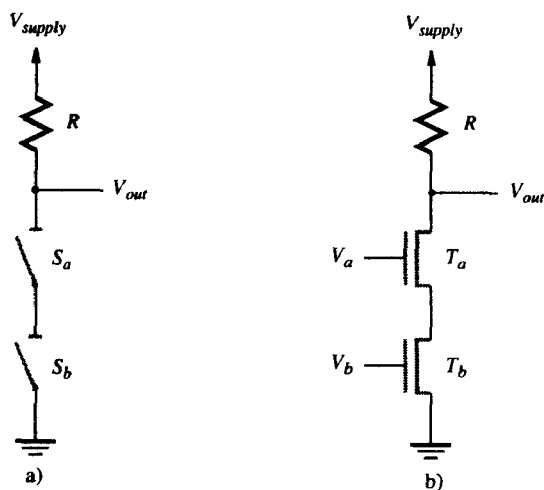
容易验证，将晶体管如图A-13那样串联，就可以得到一个与非门电路。逻辑函数“与”和“或”可以分别使用与非门和或非门，后接图A-11中的反相器实现。



图A-11 一个反相器电路



图A-12 或非门的晶体管电路实现



图A-13 与非门的晶体管电路实现

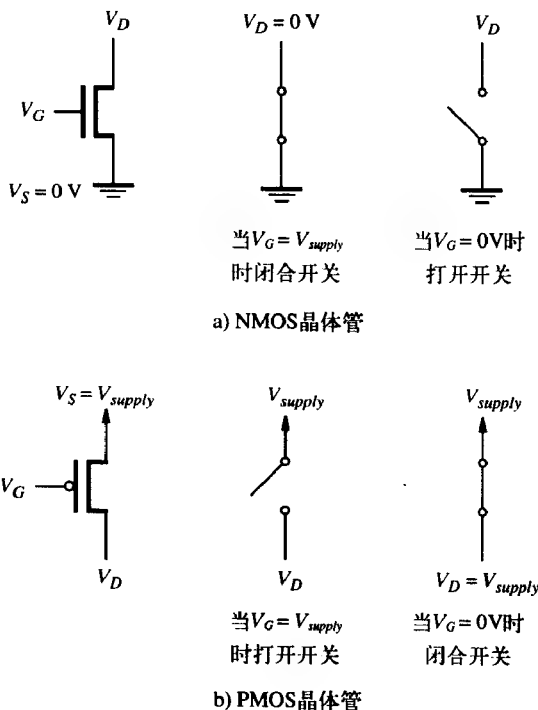
679

680

注意,与非门和或非门的电路实现比与门和或门要简单。因此,在实践中经常会使用大量与非门和或非门实现的逻辑函数。本书中所举的许多由与、或、非门组成电路的例子是为了便于理解。在实际应用中,逻辑电路包含所有这5种门电路。

### A.5.1 CMOS电路

图A-11到图A-13示例了使用NMOS技术实现电路的一般结构。这个名称起源于实现逻辑函数的晶体管是NMOS型。两种类型的金属氧化物半导体可以作为开关使用。一个 $n$ 沟道晶体管属于NMOS类型,当它的门输入上升到止的电平 $V_{supply}$ 时,表现为一个闭合的开关,如图A-14a所示。相反的表现是使用一个 $p$ 沟道的晶体管,即PMOS类型。当门电平 $V_G$ 等于 $V_{supply}$ 时,它相当于一个打开的开关,而当门电平 $V_G = 0$ 时,它相当于一个闭合的开关,如图A-14b所示。注意PMOS晶体管的图形表示在其门输入处有一个圆圈,表示它的行为与NMOS晶体管相反。还要注意PMOS晶体管的源极和漏极名称也相应与NMOS晶体管所接的端相反。NMOS晶体管的源极是接地的,而PMOS晶体管的源极与 $V_{supply}$ 相连。这些命名习惯是根据这些晶体管中的电流性质定义的。



图A-14 逻辑电路中的NMOS和PMOS晶体管

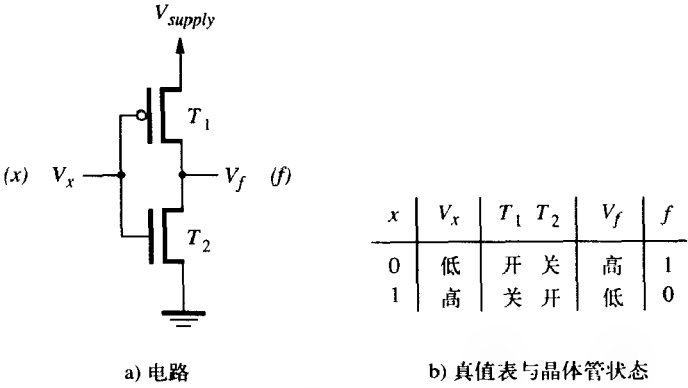
图A-11到图A-13电路的缺点在于它们的能量损耗。当开关闭合以提供地与上拉电阻 $R$ 之间的通道时,有电流从 $V_{supply}$ 流向地。在相反的状态下,即开关打开时,没有到地的通路而且也没有电流流过。(MOS晶体管的门端没有电流流过)。因此,根据门的状态,在逻辑电路中可能会出现明显的能量损耗。

解决能量损耗问题的有效方法是同时使用NMOS和PMOS晶体管来实现电路,使之在稳定状态下不消耗能量。这种方法引出了CMOS(互补金属氧化物半导体)技术。图A-15的反相器电路示例了CMOS电路的基本思想。当 $V_x = V_{supply}$ ,相应于输入 $x$ 的逻辑值为1,晶体管 $T_1$ 闭合而 $T_2$ 打开。因此 $T_2$ 将输出电平 $V_f$ 下拉到0。当 $V_x$ 变为0时,晶体管 $T_1$ 打开而 $T_2$ 闭合。这样, $T_1$ 将输出电平 $V_f$ 上拉至 $V_{supply}$ 。因此, $x$ 和 $f$ 的逻辑值互补,电路实现了一个非门。

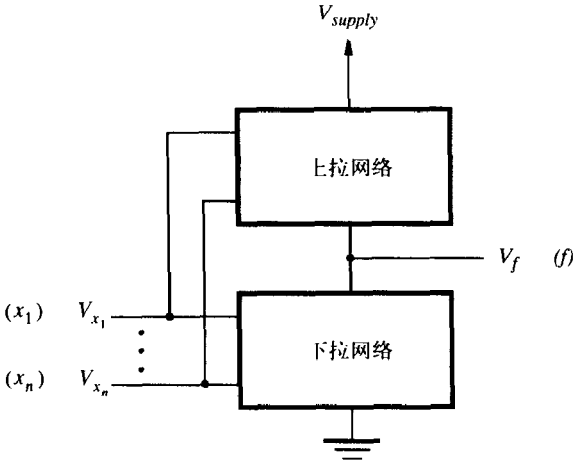
这个电路的关键特征在于晶体管 $T_1$ 和 $T_2$ 操作是完全相反的:当一个打开的,另一个就是闭合的。因此,从输出点 $f$ 到 $V_{supply}$ 或地总有一条闭合的通路。除了晶体管转换状态时非常短的过渡期外,任何时候在 $V_{supply}$ 和地之间都没有闭合通路。这表明当电路处于稳定状态时并不消耗多少能量,仅当它从一个逻辑状态转变到另一个逻辑状态时消耗能量。因此,这个电路的能量损耗是由状态转换的频率决定的。

现在我们可以将CMOS的概念扩展到 $n$ 输入电路,如图A-16所示。NMOS晶体管用来实现下拉网络,当所需函数 $F(x_1, \dots, x_n)$ 等于0时建立输出点 $f$ 和地之间的闭合通路。上拉网络由PMOS

晶体管构成，当所需函数 $F(x_1, \cdots, x_n)$ 等于1时建立输出点 $f$ 和 $V_{supply}$ 之间的闭合通路。上拉和下拉网络的功能是相反的，因此在稳定状态，输出点 $f$ 和 $V_{supply}$ 或地间只存在一条闭合通路，而不是两者同时存在。



图A-15 非门的CMOS实现



图A-16 CMOS电路结构

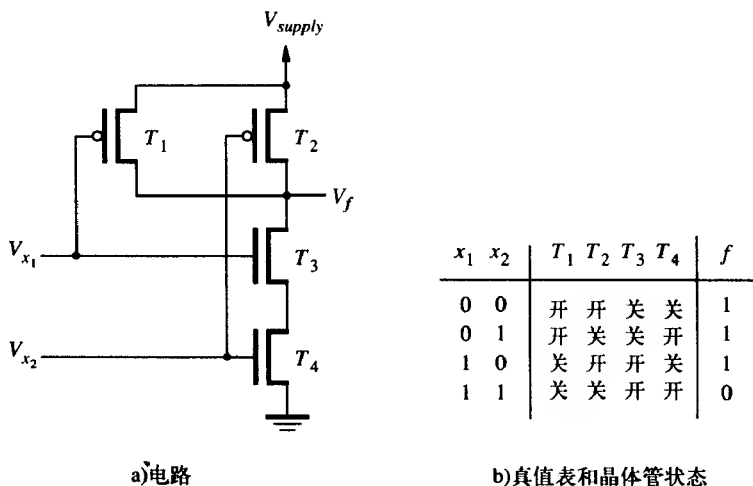
下拉网络的实现方式与图A-11至图A-13相同。图A-17给出了与非门的实现，图A-18给出了或非门的实现。图A-19通过反向与非门的输出实现了与门。

除了低能量损耗，CMOS电路的优点还有MOS晶体管体积很小，因此在集成电路芯片上只占用很小的地方。这个特性有两个显著的优势。首先，它使得在芯片上集成百万个晶体管成为可能，从而能够实现现代的微处理器和大型的存储芯片。其次，晶体管的体积越小，它从一个状态到另一个状态的转换就越快。因此，CMOS电路运行速度可在GHz范围内。

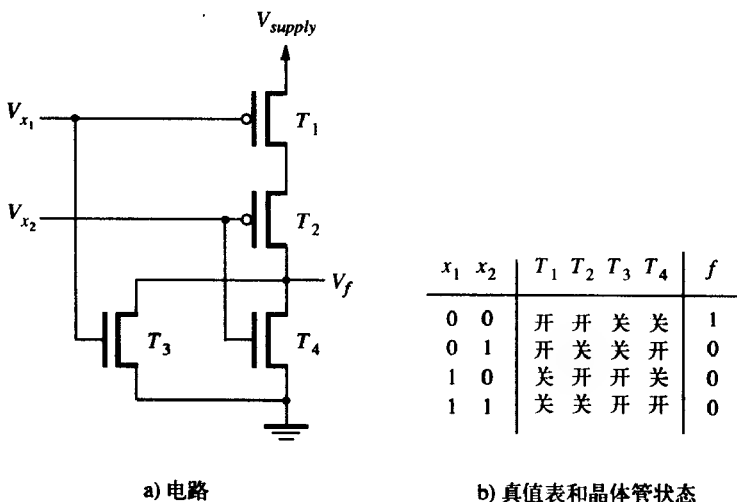
不同CMOS电路运行的电平范围在1.5~15V之间。最常用的电平值是5V和3.3V。使用低电平的电路消耗的能量更少（能量消耗约与 $V_{supply}^2$ 成正比），这意味着可以在一块芯片上放置更多的晶体管而不会过热。低电平的缺点是减小了噪声屏蔽。

CMOS反相器中高低电平信号间传输的细节如图A-20所示。粗的曲线为传输特性曲线，显示了作为输入电平函数的输出电平的变化。这一曲线表明当输入电平经过 $V_{supply}/2$ 附近时，输出电

平会有一个很陡峭的变化。这里有一个阈值电平 $V_t$ 和一个小的 $\delta$ ，如果 $V_{in} < V_t - \delta$ ，则 $V_{out} \approx V_{supply}$ ，如果 $V_{in} > V_t + \delta$ ，则 $V_{out} \approx 0$ 。这说明输出正确的信号时，输入信号不一定严格等于限定电平值0或 $V_{supply}$ 。输入信号中可能会有一些错误，即噪声，它们不会引起不利的效果。可以容忍的噪声量称为噪声容限。当输入的逻辑值为1时，这一容限是 $V_{supply} - (V_t + \delta)$ ，而当输入逻辑值是0时，这一容限是 $V_t - \delta$ 。CMOS电路具有出色的噪声容限。



图A-17 与非门的CMOS实现



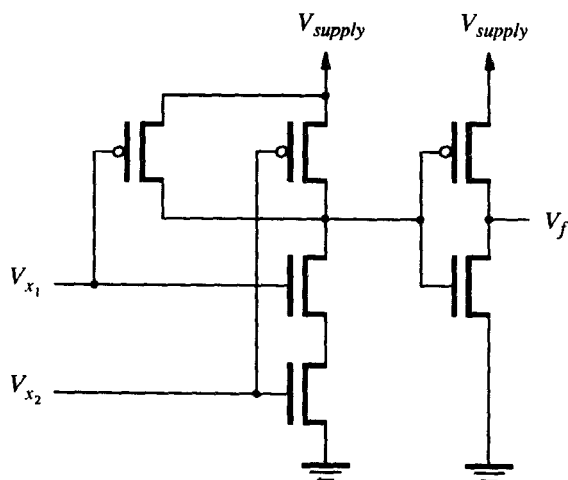
图A-18 或非门的CMOS实现

在本节中我们介绍了CMOS电路的基本特征，读者如想了解此技术的更多细节，可以查阅参考文献[1]和[8]。

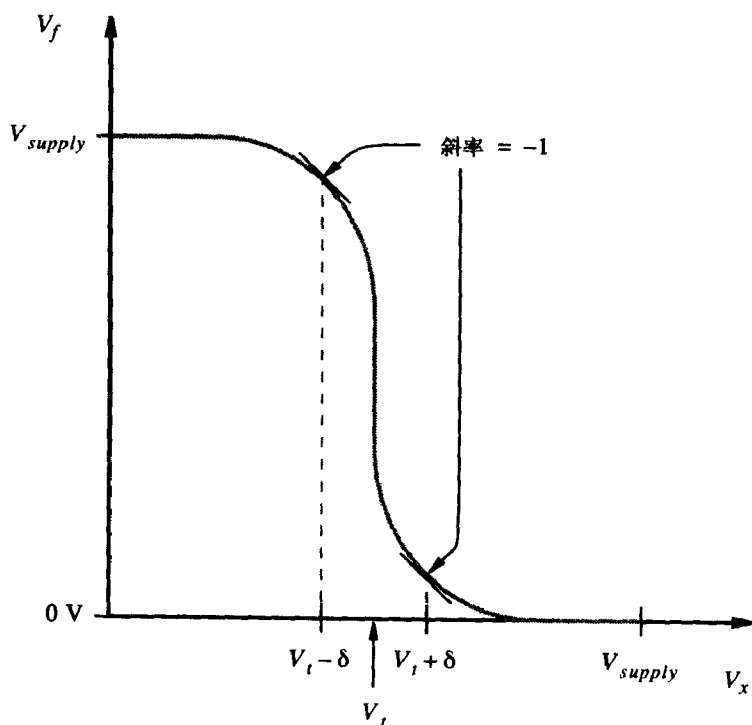
### A.5.2 传播延迟

逻辑电路不能够立即从一个状态转到另一个状态。速度由状态变化的频率来衡量。一个相

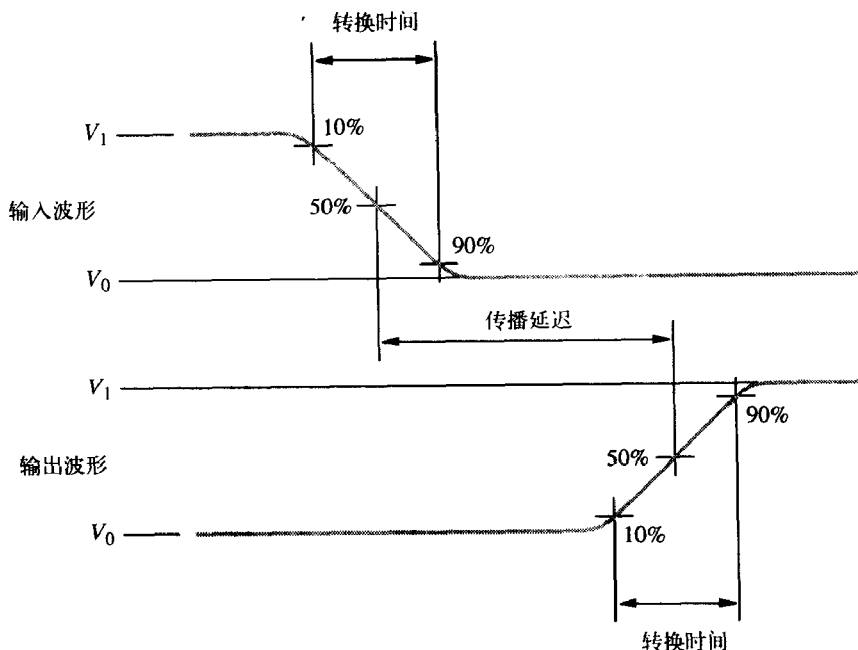
关的参数是传播延迟, 由图A-21定义。当输入的状态变化后, 在输出做出相应变化之前有一个延迟。如图所示, 通常传播延迟由两个波形转换发生50%时刻之间的时间长度衡量。另一个重要的参数是转换时间, 通常测量的是信号幅度为10%至90%点之间的时间长度, 如图所示。逻辑电路运行的最大速度随着电路通过不同路径传播延迟的增加而减少。逻辑电路中任一通路的延迟是此通路中单个门延迟的总和。



图A-19 与门的CMOS实现



图A-20 CMOS反相器的电平传输特性



图A-21 传播延迟和转换时间的定义

686

### A.5.3 扇入扇出限制

逻辑门的输入数称为扇入 (fan-in)。逻辑门输出驱动的门输入数称为扇出 (fan-out)。实际电路不允许大的扇入和扇出，因为它们都对传播延迟和电路速度产生不利的影响。

CMOS门中的每个晶体管都有一定的电容。当电容增加时，电路速度会变慢，而且信号电平和噪声容限也会变差。因此，需要对扇入和扇出进行限制，通常它是一个小于10的数。如果需要的输入数超过了最大的扇入值，必须再使用一个同类型的门。图A-9a显示了两个同类型的门如何级联。如果必须要某个门驱动的输出端数超过扇出数，可以使用两个同类型的门。

### A.5.4 三态缓冲器

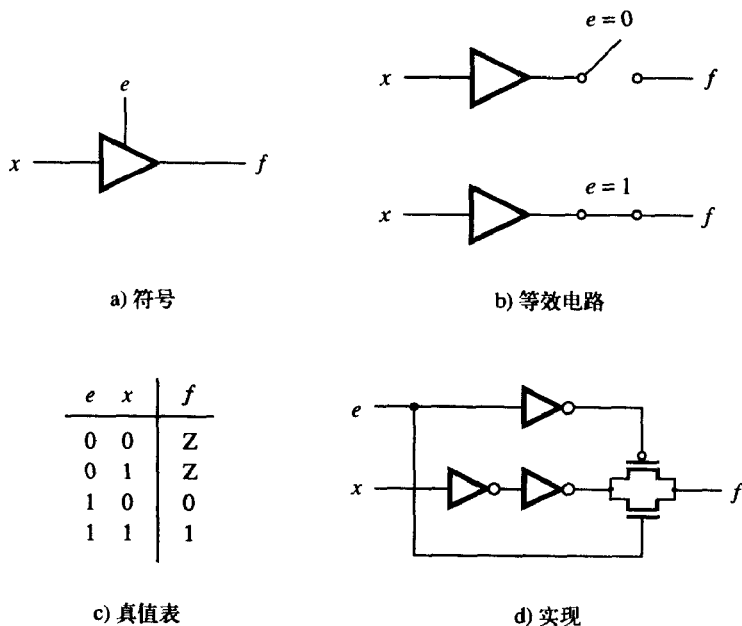
在目前讨论的逻辑门中，不能将两个门的输出连接在一起。因为当一个门的输出值是1而另一个是0时，我们不能确定组合的输出信号是什么值，这从逻辑角度讲是没有意义的。更重要的是，在CMOS电路中，输出为1的门建立了一条从输出端到 $V_{supply}$ 的直接通路，而输出为0的门建立了到地的通路。因此，这两个门会形成对电源的短路，从而对门造成损坏。

但是，在计算机系统设计中，许多时候会出现电路的输入信号可能从许多不同的源中获得的情况。这时可以使用多路复用逻辑电路，这个内容将在A.10节中讨论。也可以使用三态缓冲器的特殊门来实现。三态缓冲器具有三个状态。其中的两个状态产生普通的0和1信号。第三个状态将缓冲器的输出端置于高阻抗状态，使输出在电气上与其所驱动的输入断开。

687

图A-22描述了一个三态缓冲器。这个缓冲器有两个输入和一个输出。使能输入 $e$ 控制缓冲器的操作。当 $e = 1$ ，输出 $f$ 与输入 $x$ 具有相同的逻辑值。当 $e = 0$ ，输出被置于高阻抗状态 $Z$ 。与其等效的电路如图A-22b所示。图中三角形的符号代表一个非反相驱动器。这个电路没有实现任何逻辑操作，因为它的输出仅仅复制了输入信号，目的是提供额外的电气驱动能力。在与图中的输出

开关组合后,它的行为根据图A-22c的真值表动作。这个表描述了所需的三态行为。图A-22d 给出了三态缓冲器的电路实现。将一个NMOS晶体管和PMOS晶体管并联起来实现一个开关,与驱动的输出连接。因为这两种晶体管的门输入端需要相反的控制信号,所以要使用一个反相器。当 $e = 0$ 时,两个晶体管都关闭,相当于一个打开的开关;当 $e = 1$ 时,两个晶体管都打开,相当于一个闭合的开关。



图A-22 三态缓冲器

驱动电路必须具有能驱动大量其他门输入的能力,这些门的总容量可能会超过普通的逻辑门电路的驱动能力。为了提供足够的驱动能力,驱动电路需要更大的晶体管。因此,实现驱动器的两个级联非门所使用的晶体管要比常规逻辑门中的更大。

读者可能会疑惑为什么在输出开关中必须使用PMOS晶体管,因为从逻辑函数的角度来看,只使用NOMS晶体管也可以取得相同效果。这样做的原因是这些晶体管必须将驱动电路产生的逻辑值“传送”到输出 $f$ ,而结果表明NMOS晶体管能很好地传送0值,但对1值的效果很差,而PMOS晶体管正好相反。NMOS和PMOS的并联能够很好地传送0和1值。关于这一问题和三态缓冲器的更详细讨论,读者可以参阅参考文献[1]。

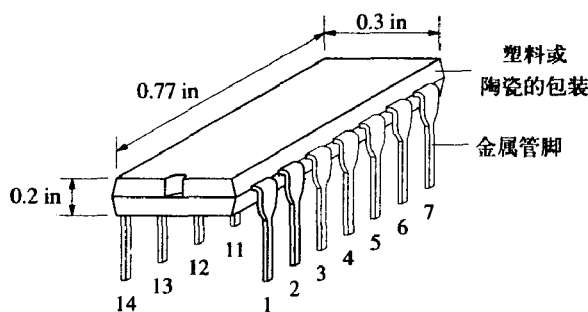
### A.5.5 集成电路封装

前面的几节讨论了实现逻辑函数的电子电路的主要特征。在实际的设计中,必须使用商业上可获得的集成电路(IC)。IC于20世纪60年代出现后,一种以标准IC芯片为形式提供逻辑门的趋势发展很快。IC芯片封装在密封的保护性外壳中,留有一些外部连接用的金属管脚。标准的IC封装有不同数目的管脚。图A-23是一个简单的包含4个与非门的封装。这4个门使用共同的电源和地线。这样仅包括几个逻辑门的IC称作小规模集成(SSI)电路。

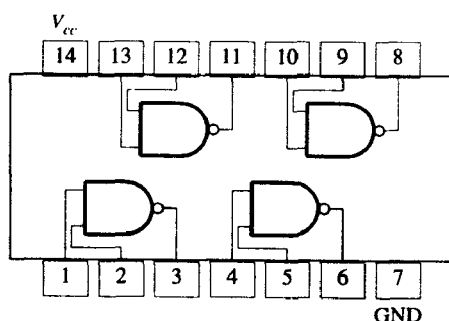
与它们需要的物理空间相比,SSI电路所提供的功能太少了。另外,IC封装上管脚的电气特征使得它们的性能也较差。一般来说,需要使用更大的晶体管提供信号,以驱动与管脚连接的



外部线路。这就增加了传播延迟和能量损耗。



a) 物理外观



b) 提供4个2输入与非门的集成电路原理图

图A-23 14管脚集成电路封装

689 图A-23中 IC封装中CMOS 与非门可能的传输延迟是5纳秒。但是，独立芯片上大型CMOS网络中的与非电路延迟可能是0.2纳秒或更短，这取决于所使用的制造工艺。

现在有许多更大的IC，而且几乎所有的逻辑电路都是用这些芯片实现的。一块芯片可能会实现一个有用的功能块，如加法器、乘法器、编码器或译码器。但是它也可能只提供一类门和可编程的互连开关，这样设计者就能用它们来实现各种各样的功能。在下面的几节中，我们将讨论一些常用的功能块和一般的用户可编程逻辑器件。

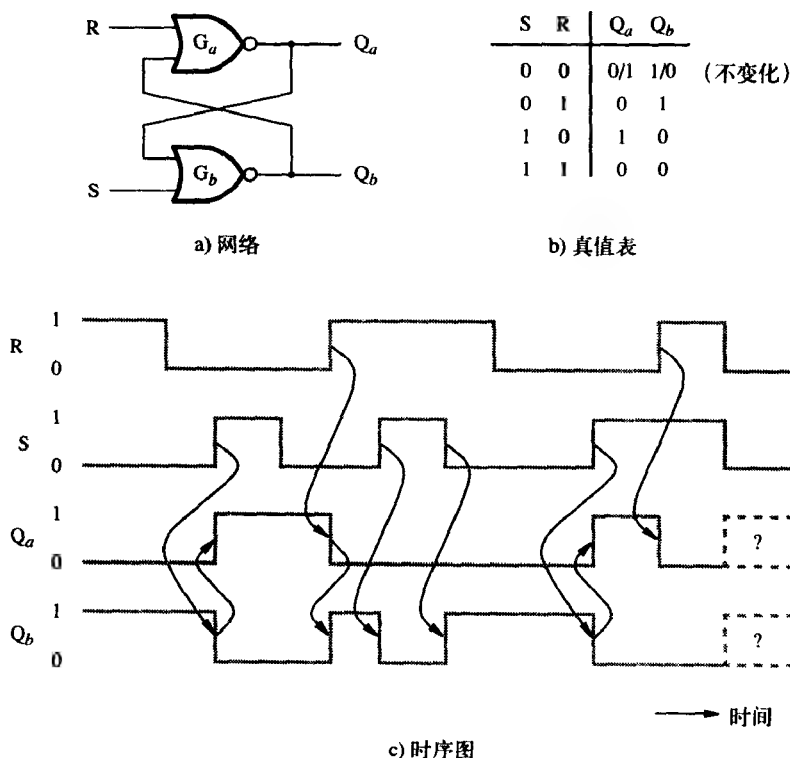
## A.6 触发器

大部分数字逻辑的应用都需要信息的存储。例如，控制密码锁的电路必须记住所拨数字的顺序，以便确定是否要打开锁。另一个重要的例子是数字计算机中的内存保存程序和数据。

存储二进制信息的基本电子元件称为锁存器。看一下图A-24a中两个交叉耦合的或非门。让我们以状态 $R = 1$ ,  $S = 0$ 开始来检测一下此电路。简单的分析表明 $Q_a = 0$ ,  $Q_b = 1$ 。在这种情况下，门 $G_a$ 的两个输入均为1。这样，如果 $R$ 变为0， $Q_a$ 和 $Q_b$ 的输出不会有任何变化。如果 $S$ 设置为1而 $R$ 等于0， $Q_a$ 和 $Q_b$ 将分别是1和0，而且将会保持这一状态直至 $S$ 变回0。因此，这个逻辑电路构成了一个存储单元或锁存器，它记下了两个输入 $S$ 和 $R$ 中哪一个最接近于1。图A-24b给出了这个锁存器的真值表。图A-24c显示了锁存器的特征波形。其中的箭头表明了信号间产生效果的关系。注意当输入 $R$ 和 $S$ 同时从1变到0时，结果状态是不确定的。实际上，锁存器将任意假设两个稳态中

的一个。输入值 $R = S = 1$ 在大多数锁存器中不使用。

根据前面电路操作的性质， $R$ 和 $S$ 线称作置位和复位输入。由于通常不使用 $R = S = 1$ ，故用 $Q$ 和 $\bar{Q}$ 来分别代替 $Q_a$ 和 $Q_b$ 。但是， $\bar{Q}$ 只是代表锁存器第二个输出的符号，而并不是 $Q$ 的反，因为输入值 $R = S = 1$ 的结果是 $Q = \bar{Q} = 0$ 。



图A-24 使用或非门实现的基本锁存器

### A.6.1 门控锁存器

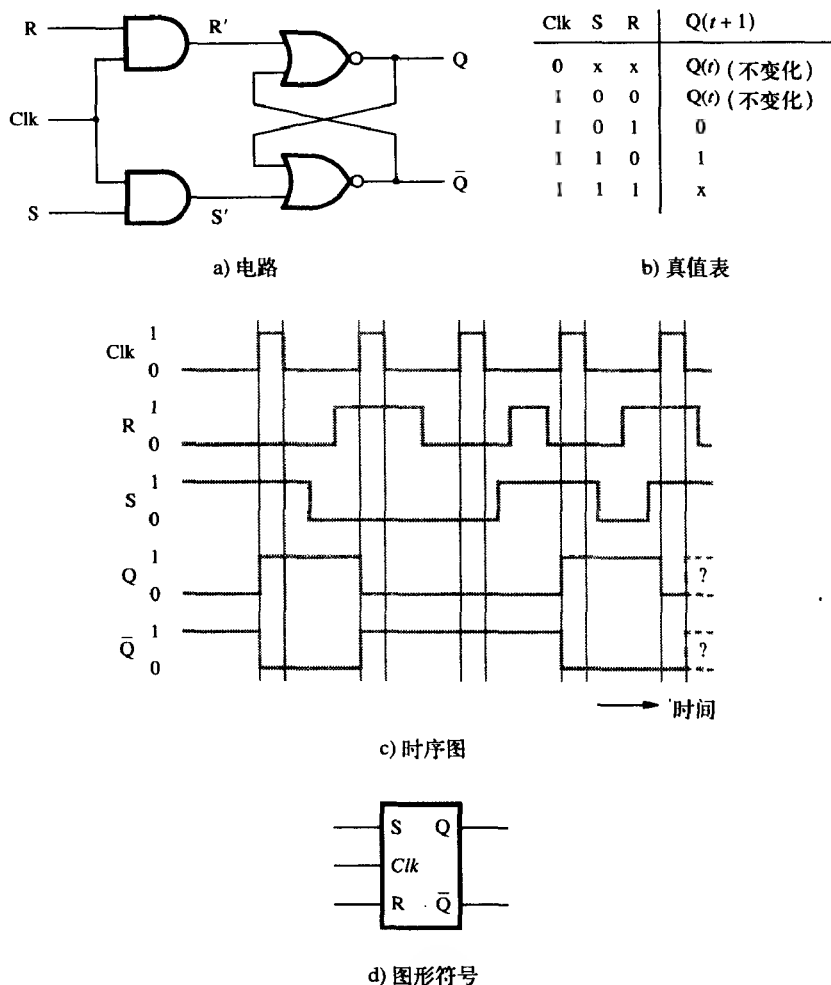
许多应用需要有 $R$ 和 $S$ 以外的输入控制锁存器置位或复位的时间，这个输入称为时钟。组合的结果称作门控SR锁存器。该锁存器的逻辑电路、真值表、波形特性和图形符号如图A-25所示。

690

当时钟 $Clk$ 等于1时， $S'$ 点和 $R'$ 点的值等于 $S$ 和 $R$ 。另一方面，当 $Clk = 0$ 时， $S'$ 点和 $R'$ 点均等于0，并且锁存器不会发生任何状态变化。

至此我们已经使用真值表描述了逻辑电路的行为。真值表给出了一个电路与各种输入值相对应的输出。每个输入值惟一地定义输出的逻辑电路为组合电路。这是在A.1到A.4节中讨论的一类电路。当出现存储元件后，我们得到了一种不同种类的电路。这种电路的输出函数不只取决于输入变量的当前值，还反映了输入变量以前的行为。图A-24给出的就是这样一个例子。这种类型的电路称为时序电路。

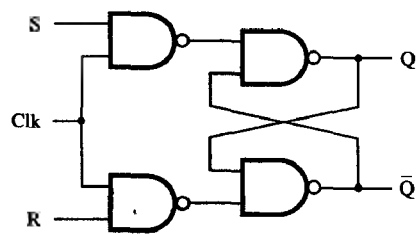
由于存储的特性，锁存器的真值表需要修改，以显示当前状态所产生的影响。图A-25b描述了门控SR触发器的行为，其中 $Q(t)$ 代表它的当前状态。到下一个状态 $Q(t+1)$ 的转换发生在一个时钟脉冲之后。注意当输入值为 $S = R = 1$ 时，由于前面讨论的原因， $Q(t+1)$ 的值是不确定的。



图A-25 门控SR锁存器

如图A-26, 门控SR锁存器可以用与非门实现。这是一个有用的例子, 可以证明这个电路在功能上等效于图A-25a中的电路 (参见习题A.20)。

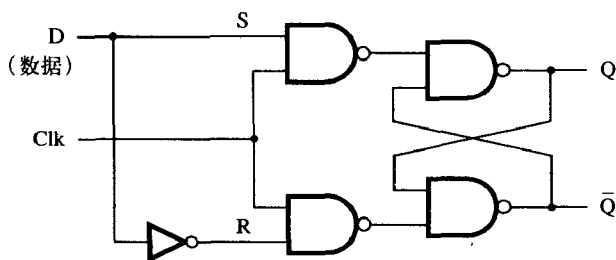
第二种类型的门控锁存器称作门控D锁存器, 如图A-27所示。在这种情况下, S和R两个信号是从单一输入D导出的。在一个时钟脉冲中, 如果 $D = 1$ , 输出Q被置位为1, 或当 $D = 0$ 输出被复位为0。D触发器当时钟为高电平时采样输入D的值并储存该值, 直到下一个时钟脉冲到来。



图A-26 使用与非门实现门控SR锁存器

### A.6.2 主从触发器

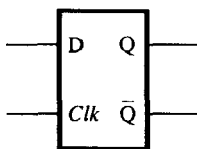
在图A-25的电路中, 假设当 $Clk = 1$ 时, 输入S和R不发生改变。考察电路会发现, 在这个时间内, 不管输入S和R发生任何变化, 输出都会立即做出响应。类似地, 在图A-27的电路中, 当 $Clk = 1$ 时,  $Q = D$ 。在许多情况下, 这种情况不是我们想要的, 尤其是对于包含计数器和移位寄存器的电路, 这些以后再讨论。在这样的电路中, 将逻辑条件立即从数据输入端 (R、S和D) 传



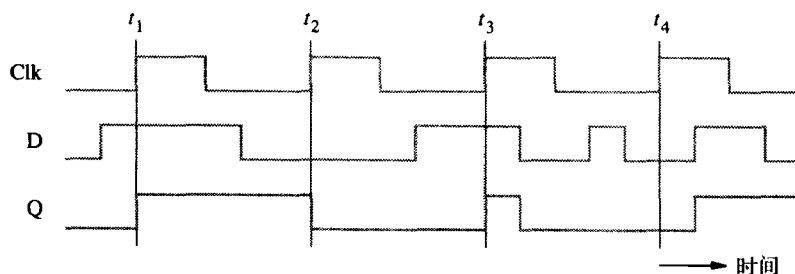
a) 电路

Clk	D	$Q(t+1)$
0	x	$Q(t)$
1	0	0
1	1	1

b) 真值表



c) 图形符号



d) 时序图

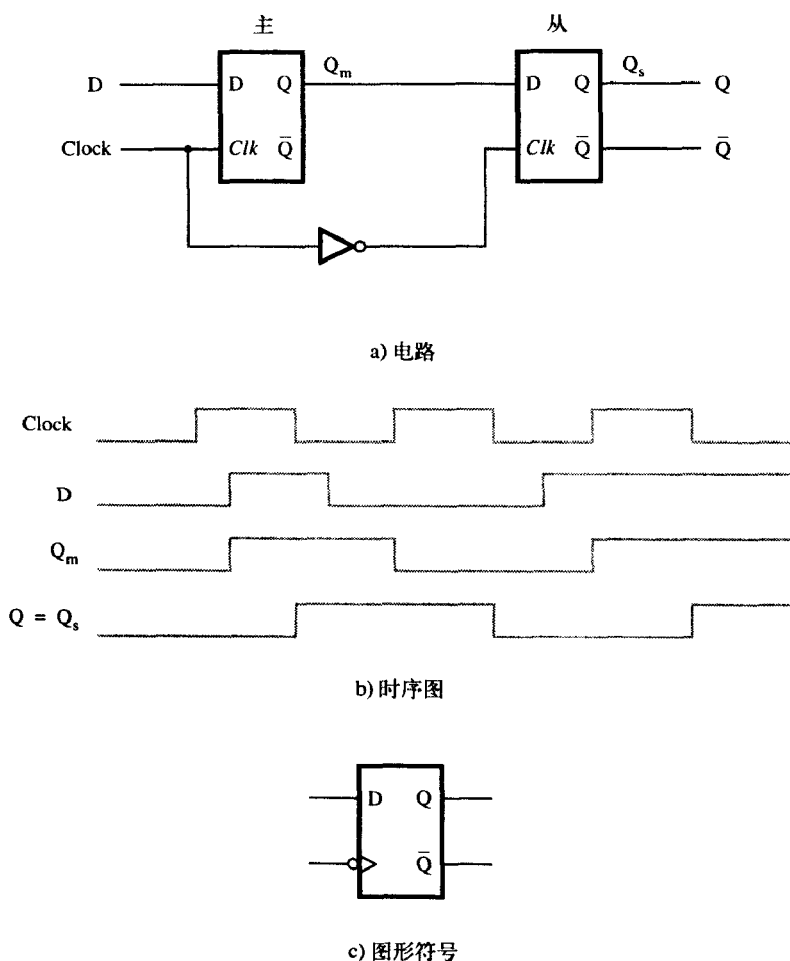
图A-27 门控D锁存器

递到锁存器的输出往往导致错误的操作。主从结构解决了这个问题。如图A-28a, 两个门控D锁存器可以连接成一个主从D触发器。首先, 看一下主触发器部分, 当Clock = 1时与输入D相连。时钟从1到0的跳变使主触发器与输入断开, 并将主触发器中存储的内容传送到从触发器。我们可以看到, 在输入D和输出Q间任何时候都不存在直接的通路。

需要注意的是, 当Clock = 1时, 主触发器的状态直接受到输入D变化的影响。从触发器的功能是在主触发器准备接收由输入D决定的下一个状态值时, 保持触发器的输出值。时钟从1到0的跳变结束后, 新状态就从主触发器传送到从触发器。这时, 主触发器已经与输入断开, 因此输入D的任何变化都不会影响传送过程。状态转换的例子如图A-28b的时序图所示。

触发器是指一种在控制时钟信号的边缘改变状态的存储元件。在前面讨论的主从D触发器中, 在时钟的下降沿(1到0)发生明显的变化。当变化达到从触发器的Q端时, 就可以观察到该变化。注意在图A-28的电路中, 也可以使用反向的时钟信号控制主触发器, 而用原时钟控制从触发器。这时, 触发器输出Q变化就会在时钟的上升沿发生。

图A-28c给出了触发器的图形符号。我们使用一个箭头来代替标签Clk定义此触发器的时钟输入。这是定义触发器上升沿触发状态变化的标准表示方法。在我们的图中是下降沿触发状态变化的, 因此在时钟输入端(箭头之外)再画一个小圆圈。



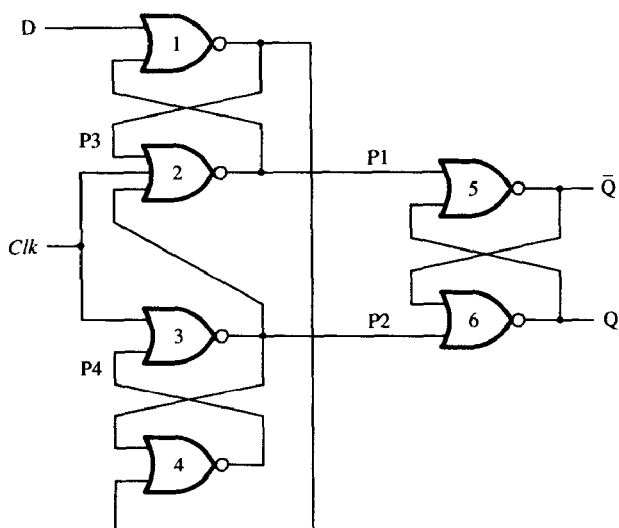
图A-28 主从D触发器

### A.6.3 边沿触发

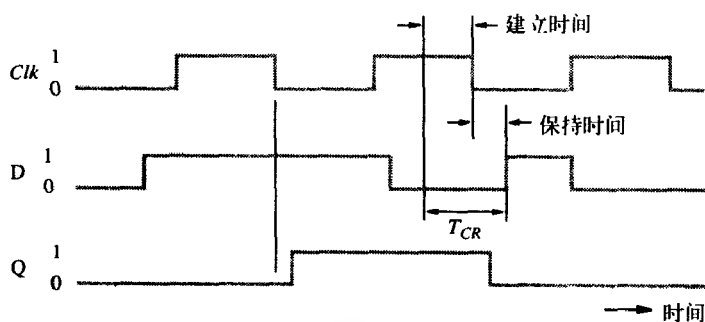
如果输入端的数据只在时钟信号跳变时传送给输出端，我们就说触发器是边沿触发的。在其他所有时间输入与输出都是断开的。上升沿（前沿）触发和下降沿（后沿）触发分别描述了数据传送发生在0到1和1到0时钟跳变的触发器。为了进行正确的操作，边沿触发的触发器要求时钟脉冲的触发沿定义明确并且跳变时间很短。图A-28中的主从触发器是一个下降沿触发器。

下降沿触发D触发器的另一种实现如图A-29a。让我们看一下这个触发器的操作过程。如果  $Clk = 1$ ，门2和3的输出均等于0。因此，触发器输出Q和  $\bar{Q}$  维持触发器的当前状态。容易验证在这段时间内，P3点和P4点会立即响应D的变化。P3点保持与  $\bar{D}$  相等，而P4点保持与D相等。当  $Clk$  下降为0时，这些值通过门2和3分别传送到P1和P2。因而，由门5和6组成的输出锁存器得到了需要储存的新状态。

我们现在验证一下当  $Clk = 0$  时，D的变化不会改变P1点和P2点。考虑两种情况。首先，假设在  $Clk$  的下降沿  $D = 0$ 。P2处的1在门2和4都保持输入为1，这使得P1和P2为0和1，与D的任何变化无关。其次，假设在  $Clk$  下降沿  $D = 1$ 。P1处的1表明D的任何变化都不会影响到门1，门1保持为0。



a) 网络



b) 时序举例

图A-29 下降沿触发的D触发器

当下一个时钟脉冲开始时 $Clk$ 上升为1，P1点和P2点再一次被强制置0，使输出与电路的其他部分断开。P3点和P4点跟随D的变化而变化，如我们前面所描述的一样。

这种D触发器的操作示例如图A-29b。触发器在 $Clk$ 从1到0跳变时所得的状态等于在这一跳变之前输入D的值。但是，在 $Clk$ 下降沿附近存在一个临界时间段 $T_{CR}$ ，其间D上的值不应改变。如图所示，这段时间分为两部分：时钟边沿前的建立时间和时钟边沿后的保持时间。时序图显示输出Q在时钟下降沿过后稍晚一些才有变化。这是由于受到了与非门传输延迟的影响。

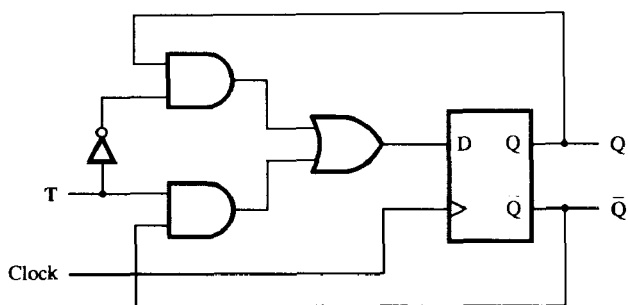
#### A.6.4 T触发器

使用最为广泛的触发器是D触发器，因为它们对数据的临时存储很有用。但是，在一些应用中，使用其他类型的触发器会很方便。在A.8节中讨论的计数器电路就是由T触发器构成的。当其输入T等于1时，T触发器在每个时钟周期都改变状态。我们称它“翻转”自己的状态。

图A-30显示了T触发器。如图A-30a所示，它的电路从D触发器得来。图中还给出了它的真值表、图形符号以及时序图举例。注意现在假设的是一个上升沿触发的触发器。

695

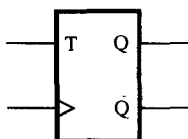
696



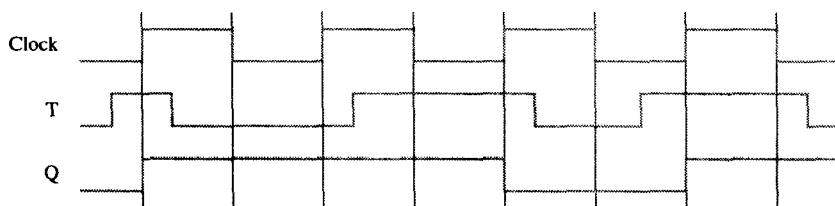
a) 电路

T	$Q(t+1)$
0	$Q(t)$
1	$\bar{Q}(t)$

b) 真值表



c) 图形符号



d) 时序图

图A-30 T触发器

### A.6.5 JK触发器

在实际中会遇到的另一种触发器是JK触发器，它组合了SR触发器和T触发器的功能，如图A-31所示。它的操作由图A-31b中给出的真值表定义。表中的前三行定义了与图A-25b（当 $Clk = 1$ 时）中一样的表现，所以J和K分别对应S和R。当输入值 $J = K = 1$ 时，下一个状态定义为与当前触发器相反的状态。即当 $J = K = 1$ ，触发器的功能就像是一个开关，翻转当前的状态。

JK触发器可以使用D触发器联结成

697

$$D = J\bar{Q} + \bar{K}Q$$

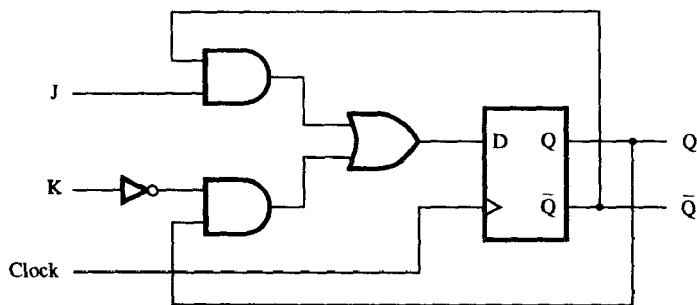
实现。相应的电路如图A-31a所示。

JK触发器可以有多种用途。它可以像D触发器一样用来存储数据。它也可以用来构造计数器，因为当J和K输入连在一起时，就表现为一个T触发器。

### A.6.6 带预置和清除的触发器

698

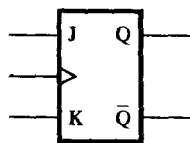
触发器的状态由它的当前状态和输入端的逻辑值决定。有时需要强制将触发器置于某个特定状态0或1，而不管它的当前状态和正常输入值是什么。例如，当打开计算机时，需要将所有的触发器都置于一个已知状态。通常是将它们的输出复位为0状态。有些时候也需要将一些触发器置为1状态。



a) 电路

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

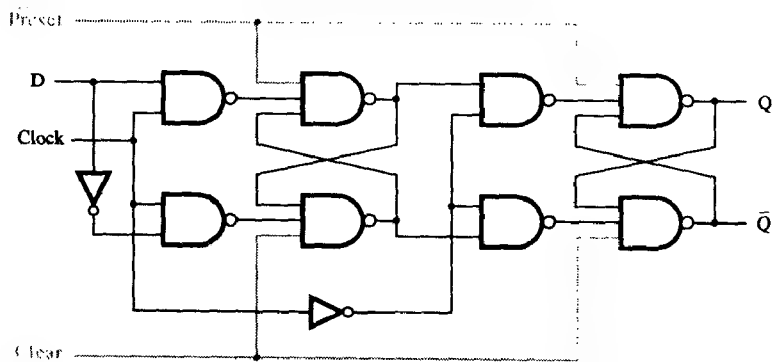
b) 真值表



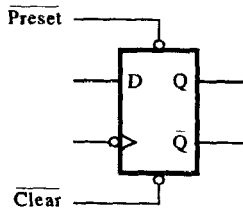
c) 图形符号

图A-31 JK触发器

图A-32显示了如何将预置和清除控制输入加到主从D触发器上，不管D输入和时钟是什么，都将触发器强制置为1或0。图中的上划线和圆圈显示这些输入是低电平有效的。当预置（ $\overline{\text{Preset}}$ ）和清除（ $\overline{\text{Clear}}$ ）输入都等于1时，触发器就在正常情况下由时钟和D输入控制。当 $\overline{\text{Preset}}$ 等于0时，触发器强置于1状态。而当 $\overline{\text{Clear}} = 0$ 时，触发器强置于0状态。在其他类型的触发器中也常常加入预置和清除控制。



a) 电路



b) 图形符号

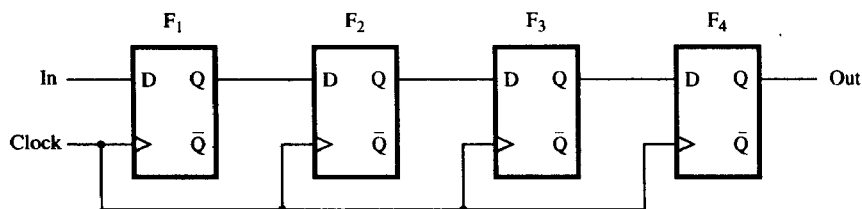
图A-32 带预置和清除的D触发器



## A.7 寄存器与移位寄存器

每个触发器可以用来存储一位数据。但是，在用字处理数据的机器中，字由许多位组成（可能是64位）。为了处理数据的方便，我们将许多触发器组合为一种常用的结构，称为寄存器。寄存器中所有触发器的操作都由共同的时钟控制。因此，数据被写入（载入）触发器或从触发器中读出都同时进行。

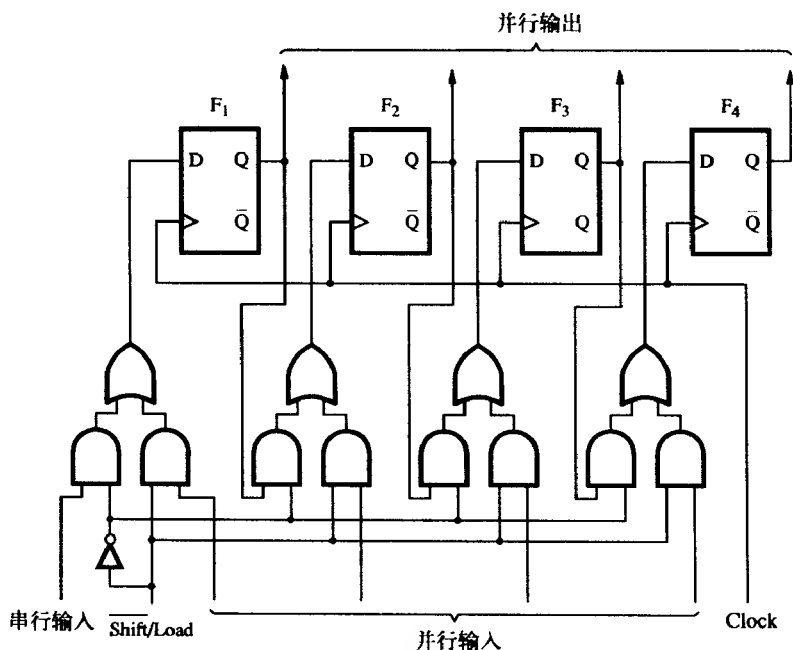
处理数字数据经常要求有移位和循环数据的能力，因此需要提供有此能力的硬件。能实现这两种操作的简单器件是寄存器，其内容可以每次向左或向右移动一位。例如图A-33中的4位移位寄存器。它是由D触发器连接成的，故每个时钟脉冲会引起从 $F_i$ 到 $F_{i+1}$ 的内容（状态）传递，表现为“右移”。数据被顺序送入或送出寄存器，将输出接至输入可以实现数据的循环。



图A-33 简单的移位寄存器

移位寄存器的正确操作要求其内容在每个时钟脉冲恰好移动一位。这就限制了可以使用的存储元件类型。图A-27描述的锁存器不适合实现此操作。当时钟为高电平时，D输入的值很快就传送到输出。接着，数据又以相同的方式经过下一个锁存器。因此，在一个时钟脉冲内不能控制移位发生的次数。这个次数由锁存器的传输延迟和时钟脉冲的持续时间决定。解决这个问题的办法是使用主从或边沿触发器。

可以并行载入和读取的移位寄存器是很实用的。这可以使用一些附加的门实现。如图A-34，



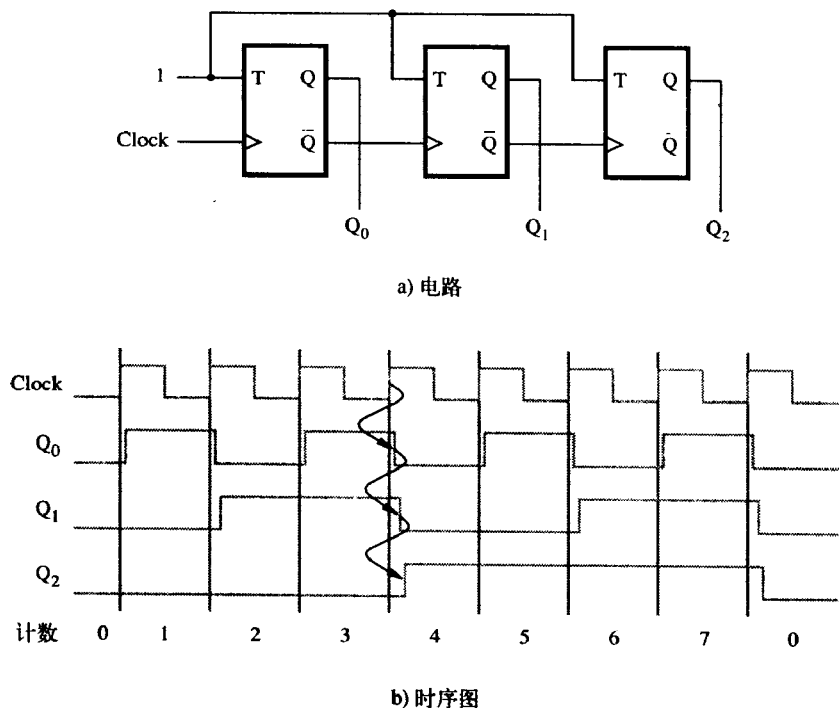
图A-34 并行访问的移位寄存器

它显示了用D触发器构成的4位寄存器。这个寄存器可以串行或并行地载入。当计时开始时, 如果  $\overline{\text{Shift}}/\text{Load} = 0$  就发生移位; 否则, 发生并行载入。

## A.8 计数器

在前面的一节中, 我们讨论了触发器在构造移位寄存器时的应用。它们在实现计数器电路时也同样有用。计数器在数字机器中的重要性无需多言。除了作为具有一般计数功能的硬件设备外, 计数器还被用来产生控制和时序信号。由高频时钟驱动的计数器可以产生频率为原始时钟分频的信号。在这样的应用中计数器作为定标器使用。

图A-35显示了由T触发器构成的一个简单的3段(或3位)计数器。回忆一下当T输入等于1时, 触发器表现为一个开关, 即每个连续的时钟脉冲都会引起状态的变化。因此, 两个时钟脉冲将会使 $Q_0$ 从1状态变为0状态再回到1状态, 或说从0到1再到0。这说明 $Q_0$ 输出波形的频率是时钟频率的一半。类似地, 因为第二个触发器是由 $Q_0$ 驱动的, 所以 $Q_1$ 的波形是 $Q_0$ 频率的一半, 或者是时钟频率的1/4。注意我们假设每个触发器的状态都在时钟输入的上升沿发生改变。



图A-35 3位开值计数器

702

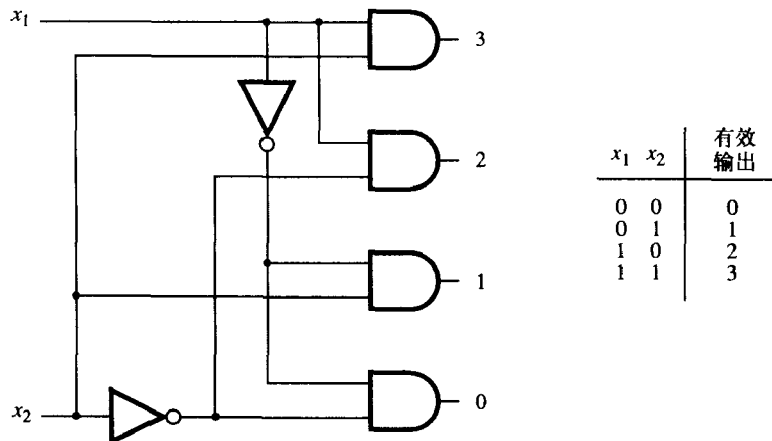
这样的计数器常被称作行波计数器, 因为输入时钟脉冲的影响像行波一样通过计数器。例如, 脉冲4的上升沿将 $Q_0$ 的状态从1变到0。 $Q_0$ 的这个变化会迫使 $Q_1$ 从1变到0, 而后按顺序迫使 $Q_2$ 从0到1。如果每个触发器产生的延迟为 $\Delta$ , 则 $Q_2$ 的延迟就是 $3\Delta$ 。当要求计数器高速运行时, 这样的延迟就会成为一个问题。但在许多应用中, 这些延迟与时钟周期相比非常小, 因而可以被忽略。

增加一些附加的逻辑门, 就可以构建一个“同步”计数器, 其每个阶段都在公用时钟控制之下, 因此所有触发器可以同时改变状态。因为总的延迟显著减少, 所以这样的计数器可以高

速运行。与之相反，图A-35的计数器被称为“异步”计数器。

## A.9 译码器

计算机中的许多信息都被保存为编码格式。在指令中，一个 $n$ 位字段可以用来表示从 $2^n$ 种可能的动作挑出一个去执行。为了执行所需的操作，编码指令必须先被译码。能够接受 $n$ 变量输入并在 $2^n$ 个输出线路中产生一个相应输出信号的电路称作译码器。图A-36给出了一个2输入4输出的简单译码器的例子。如图所示，4条输出线路由输入 $x_1$ 和 $x_2$ 选择出一个。被选出的输出具有逻辑值1，剩下的输出值都是0。



703

图A-36 2输入4输出译码器

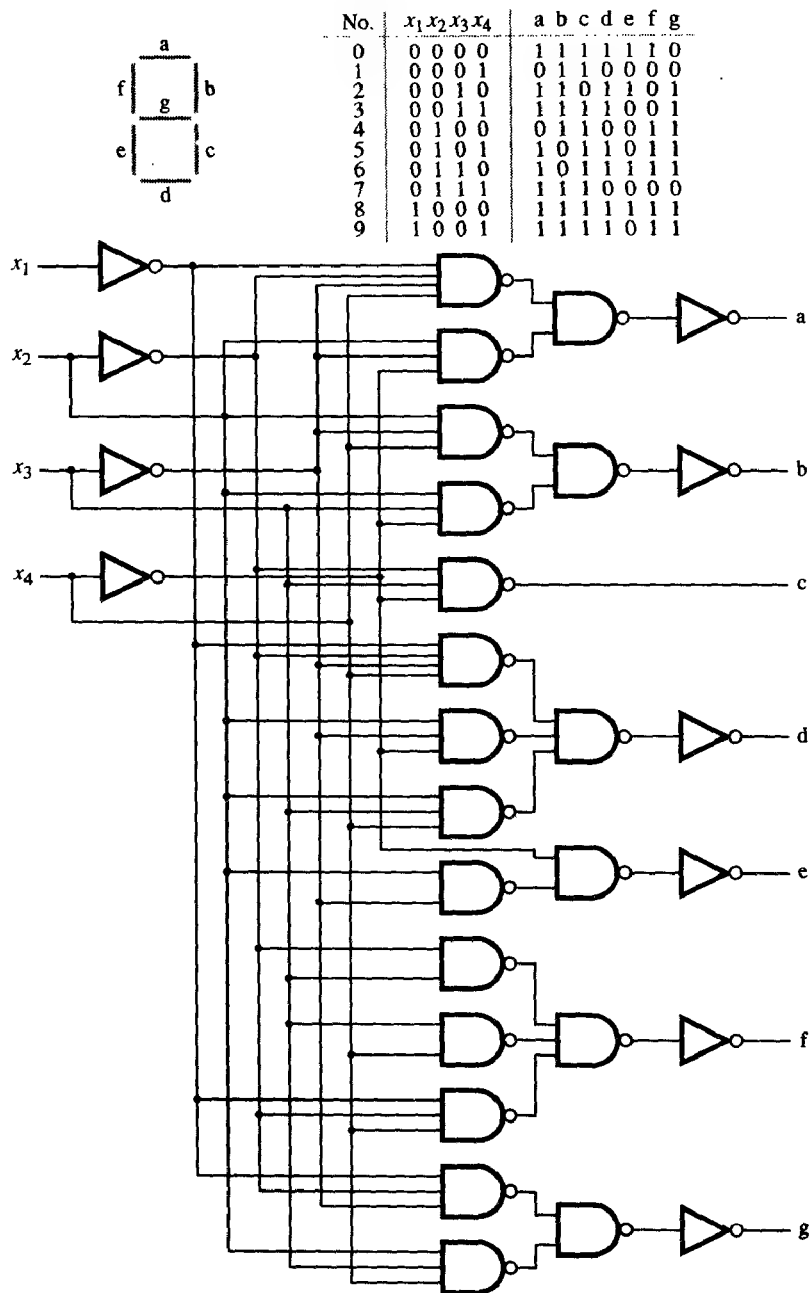
译码器还存在其他形式。例如，使用BCD码的信息经常需要一个具有4变量BCD输入的译码电路，它能从10个可能的输出中选择1个有效输出。考虑另一个具体的例子，一个能够驱动七段显示器的译码器。图A-37给出了用于显示的七段元件的结构。容易看出，从0到9的任意十进制数都可以通过打开一些段（亮）而关闭另一些段（暗）的方式显示出来。表中给出了一些必要的函数。它们可以使用图中的译码器电路实现。注意这个电路是由与非门组成的。希望读者自己验证该电路实现了所需的功能。

## A.10 多路复用器

在前面一节中，我们看到译码器根据输入信号选择一条输出线。被选中的输出线逻辑值为1，而其他输出均为0。另一种非常有用的选择电路能够从 $n$ 个数据输入中选择一个作为输出。选择操作是由一组“选择”输入控制的。这样的电路称作多路复用器。图A-38给出了一个多路复用器电路的例子。它有两个选择输入端， $w_1$ 和 $w_2$ 。它们的4个可能值用来选择4个输入 $x_1$ 、 $x_2$ 、 $x_3$ 或 $x_4$ ，作为输出 $z$ 。图中也给出了能够实现所需操作的简单逻辑电路。显然，相同的结构可以实现更大的多路复用器，用 $k$ 个选择输入端将 $2^k$ 个数据输入端中的一个连接到输出。

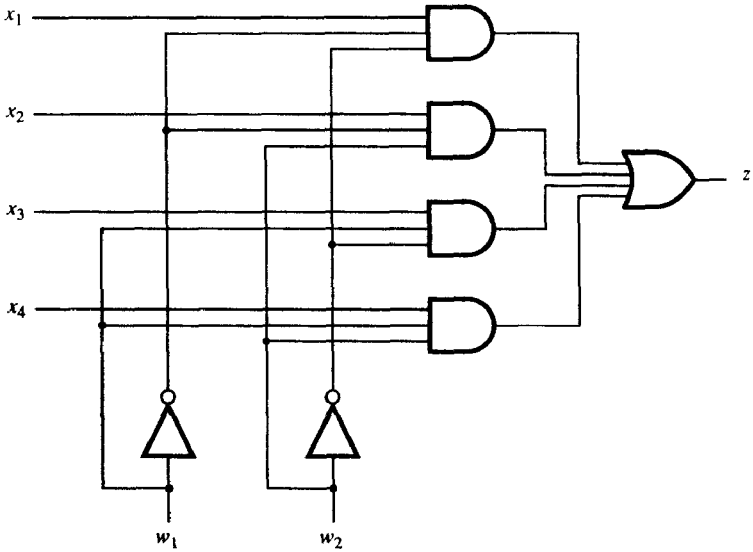
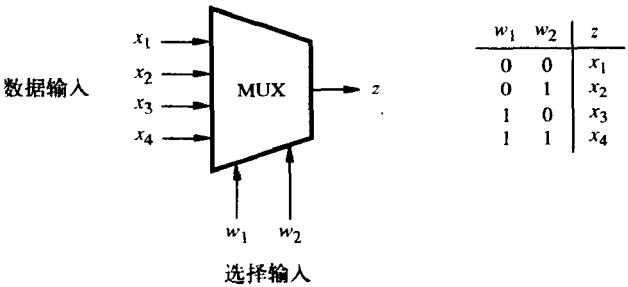
多路复用器的一个常见应用是筛选可能来自许多不同源的数据。例如，从4个数据源加载一个16位数据寄存器可以用16个4输入多路复用器实现。

多路复用器也可作为实用的基本元件用来实现逻辑函数。考虑由图A-39的真值表定义的函



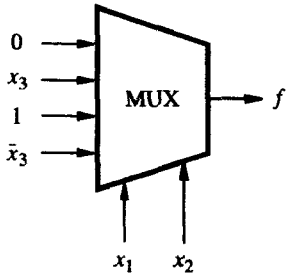
图A-37 BCD码七段显示器译码器

数 $f$ 。如图，它可以用提取的变量 $x_1$ 和 $x_2$ 代表。注意对 $x_1$ 和 $x_2$ 的每个值，函数 $f$ 对应于4项中的一个： $0$ 、 $1$ 、 $x_3$ 或 $\bar{x}_3$ 。这意味着可以使用一个4输入多路复用器电路，其中 $x_1$ 、 $x_2$ 是选择4个数据输入之一的选择输入端。这样，如果将 $0$ 、 $1$ 、 $x_3$ 和 $\bar{x}_3$ 按照真值表的要求连接到数据输入，则多路复用器的输出就是函数 $f$ 。这种方法完全是通用的。任何3变量函数都可以由一个4输入多路复用器实现。类似地，任何4变量的函数都可以由一个8输入多路复用器实现，依次类推。



图A-38 4输入多路复用器

$x_1$	$x_2$	$x_3$	$f$	$\Rightarrow$	$x_1$	$x_2$	$f$
0	0	0	0		0	0	0
0	0	1	0				
0	1	0	0		0	1	$x_3$
0	1	1	1				
1	0	0	1		1	0	1
1	0	1	1				
1	1	0	1		1	1	$\bar{x}_3$
1	1	1	0				

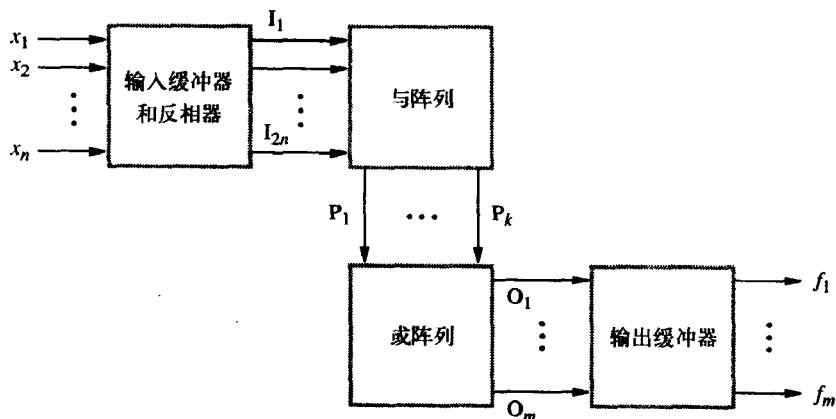


图A-39 用多路复用器实现逻辑函数

## A.11 可编程逻辑器件 (PLD)

A.2节和A.3节表明了给定的开关函数可以由积之和表达式表示,并可以使用相应的与或门网络实现。A.10节介绍了如何使用多路复用器实现开关函数。在这一节中,我们将要考虑另外一种能实现同样功能的电路。这些电路由一系列开关元件构成,可以通过编程实现积之和表达式,它们称作可编程逻辑器件(PLD)。

图A-40给出了PLD的框图。它有 $n$ 个输入变量( $x_1, \dots, x_n$ )和 $m$ 个输出函数( $f_1, \dots, f_m$ )。每个函数 $f_i$ 都由包含输入变量的积之和项组成。变量 $x_1, \dots, x_n$ 的真值或反值形式送入与阵列,形成 $k$ 个乘积项。然后这些乘积项被送入或阵列,形成输出函数。两种普遍使用的PLD类型在本节后面讨论。



图A-40 PLD的框图

### A.11.1 可编程逻辑阵列 (PLA)

如果电路同与阵列、或阵列的连接是可编程的,则该电路称为可编程逻辑阵列(PLA)。图A-41用简单的例子说明了PLA的功能结构。当可编程连接没有同与门给出的输入相连时,该输入会表现为好像是逻辑1在驱动它一样(该输入对这个门实现的乘积项不起作用)。类似地,如果没有同或门给出的输入相连时,该输入对这个门的输出没有任何影响(该输入会表现为好像是逻辑0在驱动它一样)。

可编程连接可以用不同的方法实现。一种方法是,熔断不需连接位置的金属丝。这需要使用高于常值的电流。另一种可行方法是使用可擦除存储元件(参见5.3节的EPROM存储电路)控制的晶体管开关来提供所需的连接。这也使得PLA可以重新编程。

图A-41中简单的PLA可以从3个输入变量生成4个乘积项。使用这些乘积项可以实现两个输出函数。有些乘积项可以被不止一个输出函数使用。PLA实现了下面两个函数:

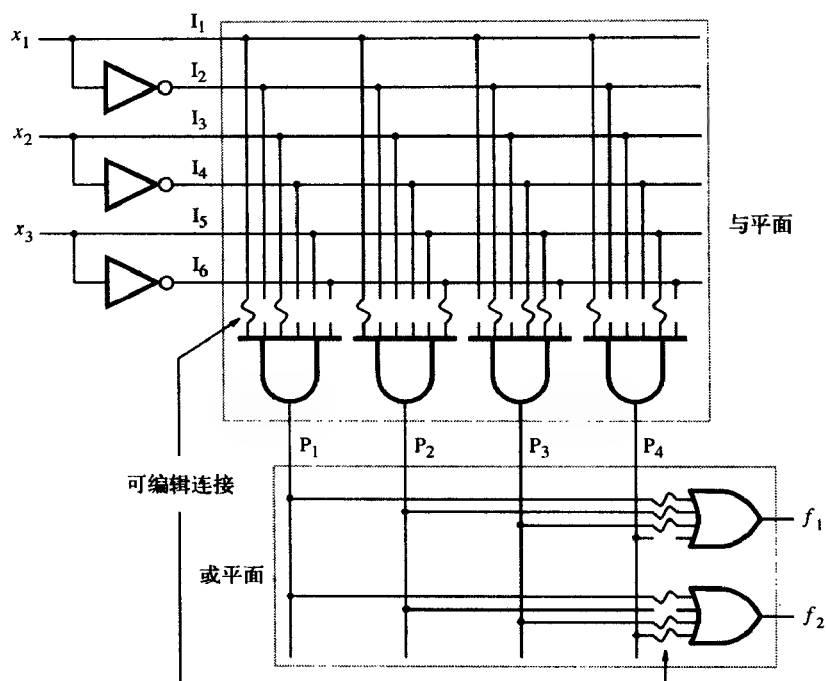
$$f_1 = x_1 x_2 + x_1 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3$$

$$f_2 = x_1 x_2 + x_1 x_3 + \bar{x}_1 \bar{x}_2 x_3$$

在两个函数中有两个相同项,因此只需要4个乘积项。实际应用中的PLA规模要大得多。

尽管图A-41清楚地描述了PLA的基本功能,但这种表示形式对于更大的PLA就显得有些笨拙了。在科技文献中已经形成一个惯例,使用只有一条输入线的相应门符号来表示乘积项和求和项。对于每个已编程连接在该线上打一个叉号“×”。图A-42使用这种图示方法表示了图A-41中

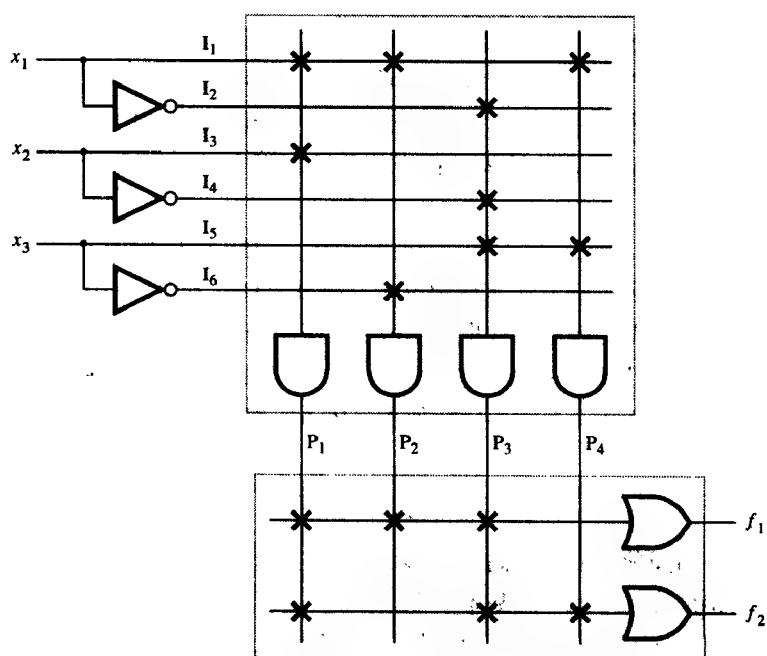
PLA的例子。通常, 为了实现输入变量的任意函数, 图中任何垂直线和水平线的交点都可以是一个可编程的连接。



$$f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$$

$$f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$$

图A-41 PLA的功能结构



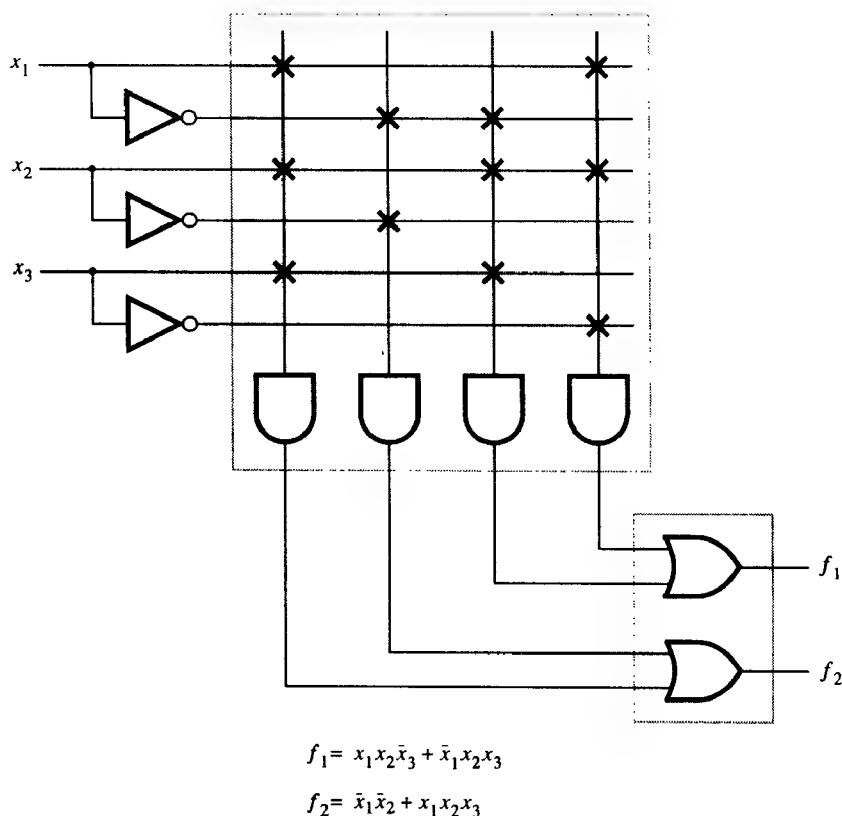
图A-42 图A-41中PLA的简化图

就其在集成电路芯片上实现所需的面积而言,PLA的结构是十分高效的。出于这种原因,在处理器芯片上经常使用这样的结构来实现控制电路。这时,制造过程的最后一步就是做好所需的连接,而制造出芯片后再使它们可编程。

### A.11.2 可编程阵列逻辑 (PAL)

在PAL中,与阵列、或阵列的输入都是可编程的。在实际应用中大量应用了一种相似设备,与阵列的输入是可编程但同或门的连接是固定的。这样的设备称作可编程阵列逻辑(PAL)芯片。

图A-43给出了实现两个函数的PAL的简单例子。每个或门所连接的与门数量决定了给定函数积之和表达式中乘积项的最大数目。与门与特定的或门永久地保持连接,这意味着在输出函数中不能共享乘积项。



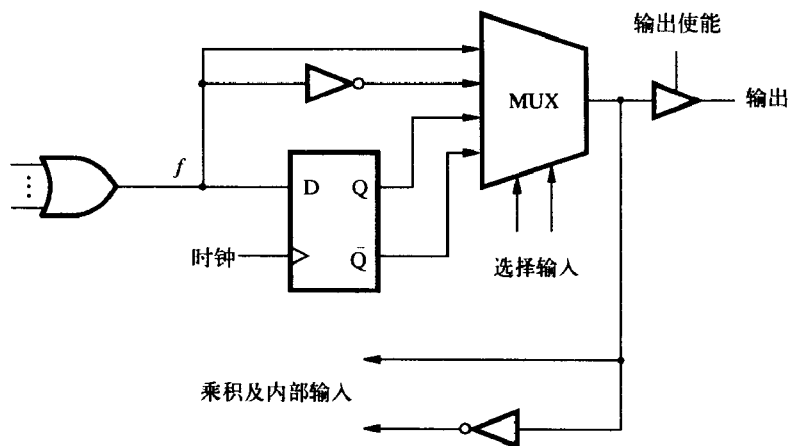
710

图A-43 PAL举例

PAL芯片在许多结构中都可使用。提供足够数量的输入变量和输出函数,就能实现大规模的函数。或门的输出若连接触发器,会使PAL具有更多功能。这样的PAL芯片使数字系统设计师能够使用一块芯片实现比较复杂的逻辑网络。

图A-44显示了电路的灵活性。多路复用器用来选择在PAL芯片输出管脚表示的 $f$ 是真值、反值还是储存值(从前一个时钟周期得出)。多路复用器的选择输入可以设置为可编程的连接。输出管脚在输出使能信号控制下由三态驱动器驱动。注意多路复用器的输出信号还可作为内部输入,用来对PAL中的其他或门提供乘积项。这使得具有多个层次(阶段)逻辑门的电路容易实现。





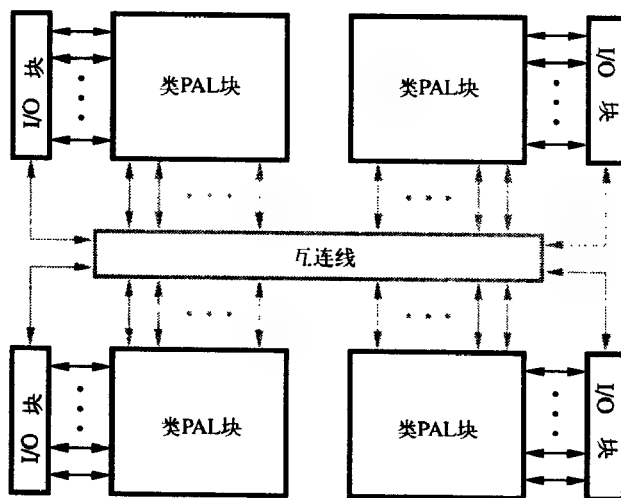
图A-44 PAL元件输出举例

### A.11.3 复杂可编程逻辑器件 (CPLD)

PAL 是很有用的器件，但是它们比较小，这意味着要实现一个典型的数字系统需要很多这样的芯片。为了解决这一问题，人们开发出了相似类型的较大器件。这就是复杂可编程逻辑器件 (CPLD)。这些器件包括两个或更多的类PAL块和可编程互连线。图A-45给出了CPLD 芯片的结构。每个类PAL块都与许多输入/输出管脚相连。类PAL块的连接通过对与互连线相关联的开关编程来建立。

互连线由水平线和垂直线构成。每条水平线都可通过对相应开关的编程与几条垂直线连接。提供完全连接，即每条水平线可与任意的垂直线连接，是不可行的，因为所需的开关数会很大。少量的开关即可实现满足需求的连通度。

市场上的CPLD尺寸不同，从2个到多于100个类PAL块都有。通过向JTAG端口加载编程信息可以对CPLD芯片编程。它是一个4管脚端口，遵循由联合测试工作组 (Joint Test Action Group) 制定的IEEE标准。

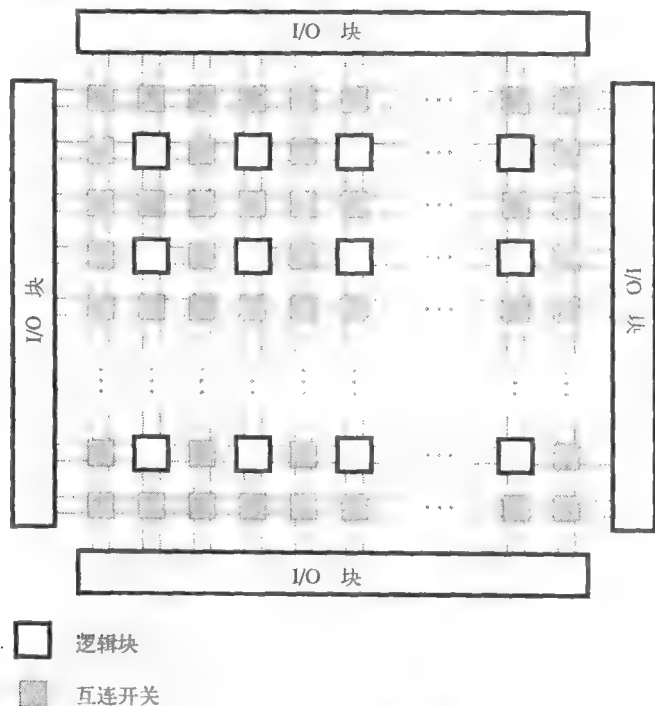


图A-45 复杂可编程逻辑器件 (CPLD) 的结构

## A.12 现场可编程门阵列 (FPGA)

PAL芯片提供了基本的功能,但是在尺寸上有所限制,因为每个积之和电路都需要一个输出管脚。为此开发出了一类功能更强大的可编程器件来克服尺寸的局限。这就是现场可编程门阵列(FPGA)。图A-46给出了FPGA的概念框图。它由一个逻辑块阵列(表示为较大的黑框)组成,这些逻辑块可以由通用互连资源连接。互连开关表示为较小的方块,由导线和可编程开关组成。这些开关用来连接逻辑块和导线,以及在不同导线之间建立所需的连接。这给予了芯片很大的路由灵活性。对芯片管脚的访问提供了输入和输出缓冲器。

712



图A-46 FPGA的概念框图

逻辑块和互连结构的设计是多种多样的。逻辑块可能只是A.10节中简单的基于多路复用器的电路,可以实现逻辑函数。另一种流行的设计是将简单的查找表作为逻辑块。例如,一个4输入查找表可以用16位存储电路实现,该电路储存逻辑函数的真值表。每个存储位与一个输入变量的组合相对应。对这样的查找表编程可以实现4变量的任何函数。逻辑块可能会包含触发器,以提供类似图A-44中额外的灵活性。

除了逻辑块,许多FPGA芯片还包含相当数量的存储单元(图A-46中没有显示),可以实现诸如先进先出(FIFO)队列或片上系统的RAM和ROM组件等结构,这些在第9章进行了讨论。

713

从使用者的角度来看,FPGA和CPLD有两个主要的不同点。FPGA的功能更为强大,可以用来实现相当大的逻辑网络。一块FPGA芯片可以实现需要超过百万个逻辑门的电路。第二个重要的考虑是这些设备的速度。因为使用可编程开关来建立互连中的所有连接,与灵活性较小的器件如PAL或CPLD相比,FPGA不可避免地具有明显的传输延迟。

FPGA的不断普及是因为它允许设计者在一块芯片上实现非常复杂的逻辑网络,而无需设计

制造定制的VLSI芯片, 这些芯片非常昂贵而且费时。使用CAD工具可以在几天内完成FPGA的设计, 而不是像制造定制VLSI芯片那样需要花费几个月的时间。FPGA实现的成本也很吸引人。即使是最大型的FPGA也仅仅耗费几百美元, 而与设计时间相关的成本和设计定制芯片相比是非常小的。

关于可编程逻辑器件的介绍性讨论可以在许多关于逻辑设计的书籍中找到。如果要对这些器件有更多的了解, 读者可以查阅其他文献[1、3~6]和制造商手册。

## A.13 时序电路

组合电路的输出完全由当前的输入决定。A.9节和A.10节给出的译码器和多路复用器就是组合电路的例子。另一类电路的输出是由当前的输入和以前的输入序列共同决定。它们称为时序电路。这样的电路可以处于不同的状态, 这些状态由在给定时限内的输入序列决定。电路的状态决定了电路在不同输入模式下的行为。在A.7节和A.8节中, 我们曾经遇到这种电路的两种具体形式: 移位寄存器和计数器。本节将介绍更多时序电路的例子, 并给出这些电路的一般形式和设计这类电路的简短介绍。

### A.13.1 升值/贬值计数器实例

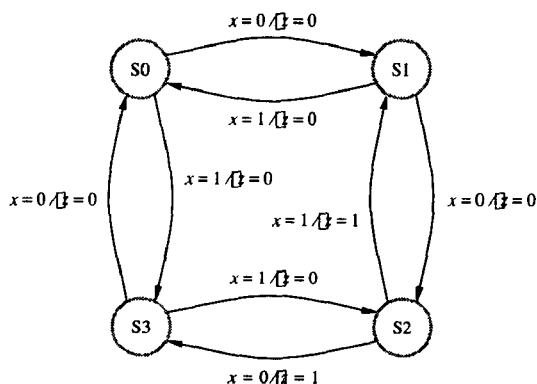
图A-35给出了由三个T触发器实现的开值计数器的结构, 按照0, 1, 2, ..., 7, 0, ...的顺序计数。也可以用类似的电路实现贬值计数, 即按0, 7, 6, ..., 1, 0, ...计数(见习题A.26)。这些简单的电路都利用了T触发器的翻转性质。

下面考虑一下使用D触发器实现计数器的可能性。作为一个具体的例子, 我们将设计一个既能使用升值计数也能使用贬值计数的计数器, 升值或贬值由一个外部控制输入的值决定。为了使这个例子简单一点, 我们将计数器限制为模4计数器, 它只需要两个状态位来表示四个可能的计数值。下面将说明如何使用构造时序电路的基本方法来设计这个计数器。所需电路在输入信号 $x$ 为0时进行升值计数,  $x$ 为1时进行贬值计数。计数发生在时钟信号的下降沿。假设我们对计数为2时的状态感兴趣。这样, 输出信号 $z$ 在计数为2时为1, 而在其他时刻都为0。

所需的计数器可以作为时序电路实现。当一个时钟脉冲到来时, 为了确定新的计数, 知道 $x$ 的值和当前的计数值就够了。我们无需知道先前输入值的实际顺序, 而只需知道当前的计数值。这个计数值决定了电路的当前状态, 这是电路保存的关于先前输入的惟一信息。如果现在的计数是2并且 $x=0$ , 下一个计数就是3。从3贬值计数或从1升值计数到2没有任何区别。

在给出电路实现之前, 先用状态图描述计数器所需的行为。计数器具有4个不同的状态:  $S_0$ ,  $S_1$ ,  $S_2$ 和 $S_3$ 。状态图是用圆圈(有时称为节点)表示状态的图。状态之间的转换用带标记的箭头表示。与箭头对应的标记指示了会使特定转换发生的输入 $x$ 的值以及产生输出结果的值。图A-47显示了这个升值/贬值计数器的状态图。例如, 从状态 $S_1$ (计数值=1)发出的箭头在输入 $x=0$ 时指向状态 $S_2$ , 这样指示了向状态 $S_2$ 的转变。它还表明当电路处于 $S_1$ 状态且 $x$ 的值为0时输出 $z$ 必等于0。从 $S_2$ 到 $S_3$ 的箭头表明当 $x=0$ 时, 下一个时钟脉冲会发生从状态 $S_2$ 到状态 $S_3$ 的转变, 并且在电路在状态 $S_2$ 时, 输出 $z$ 必是1。

注意, 状态图描述了计数器的功能行为, 而并没有提及它如何实现。图A-47可以用来描述按这种方式表现的电子数字电路、机械计数器或者计算机程序。状态图是描述具有时序行为的任何系统的有力工具。



图A-47 检测计数2的模4升值/降值计数器状态图

当前状态	下一状态		输出z	
	x = 0	x = 1	x = 0	x = 1
S0	S1	S3	0	0
S1	S2	S0	0	0
S2	S3	S1	1	1
S3	S0	S2	0	0

图A-48 升值/降值计数器例子的状态表

表示状态图信息的另一种方法是使用状态表。图A-48给出了图A-47例子的状态表。表中显示了在输入x下，从所有当前状态到下一状态的转换。输出信号z由电路的当前状态和输入x的值决定。

715

我们已经描述了一般条件下的升值/降值计数器，现在考虑它的物理实现。需要两个位对表示计数值的4个状态进行编码。令这两个位为 $y_2$ （高位）和 $y_1$ （低位）。计数器的状态由 $y_2$ 和 $y_1$ 的值决定，我们将它写成 $y_2 y_1$ 的形式并给 $y_2 y_1$ 赋值为： $S0 = 00$ ， $S1 = 01$ ， $S2 = 10$ 和 $S3 = 11$ 。这样安排使得二进制数 $y_2 y_1$ 可以明显地表示计数值。变量 $y_2 y_1$ 称为时序电路的状态变量。使用这一状态分配，图A-49给出了我们所举例子的状态表。注意使用变量 $Y_1$ 和 $Y_2$ 来表示下一状态，它们的用法与 $y_1$ 、 $y_2$ 一样。

需要注意的很重要的一点是，我们可以选择另一种 $y_2 y_1$ 对不同状态的赋值方式。例如，可能赋值为 $S0 = 10$ ， $S1 = 11$ ， $S2 = 01$ ， $S3 = 00$ 。对于一个计数器电路来说，这种赋值方式没有图A-49直观，但是结果电路仍然会运行正常。通常，使用不同状态赋值方式实现电路的成本也不相同（见习题A.32）。

我们举这个例子的目的是使用D 触发器存储连续时钟脉冲间的两个状态变量的值。触发器的输出Q是当前状态变量 $y_i$ ，输入D是下一状态变量 $Y_i$ 。注意 $Y_i$ 是 $y_2$ 、 $y_1$ 和x的函数，如图A-49所示。从图中我们可以看到

当前状态	下一状态		输出z	
	x = 0	x = 1	x = 0	x = 1
$y_2 y_1$	$Y_2 Y_1$	$Y_2 Y_1$		
0 0	0 1	1 1	0	0
0 1	1 0	0 0	0	0
1 0	1 1	0 1	1	1
1 1	0 0	1 0	0	0

图A-49 图A-48例子的状态分配

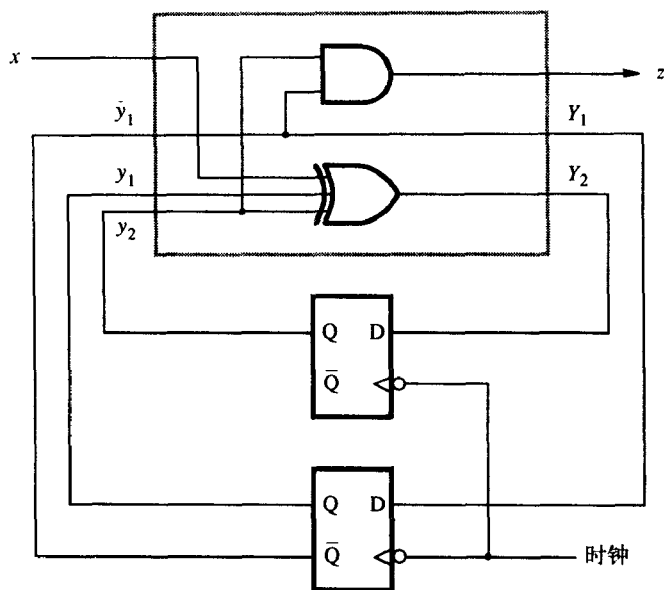
$$\begin{aligned}
 Y_2 &= \bar{y}_2 \bar{y}_1 \bar{x} + y_2 \bar{y}_1 \bar{x} + \bar{y}_2 \bar{y}_1 x + y_2 y_1 x \\
 &= y_2 \oplus y_1 \oplus x \\
 Y_1 &= \bar{y}_2 \bar{y}_1 \bar{x} + y_2 \bar{y}_1 \bar{x} + \bar{y}_2 \bar{y}_1 x + y_2 \bar{y}_1 x \\
 &= \bar{y}_1
 \end{aligned}$$

716

输出z由式子

$$z = y_2 \bar{y}_1$$

决定。这些表达式形成了图A-50所显示的电路。

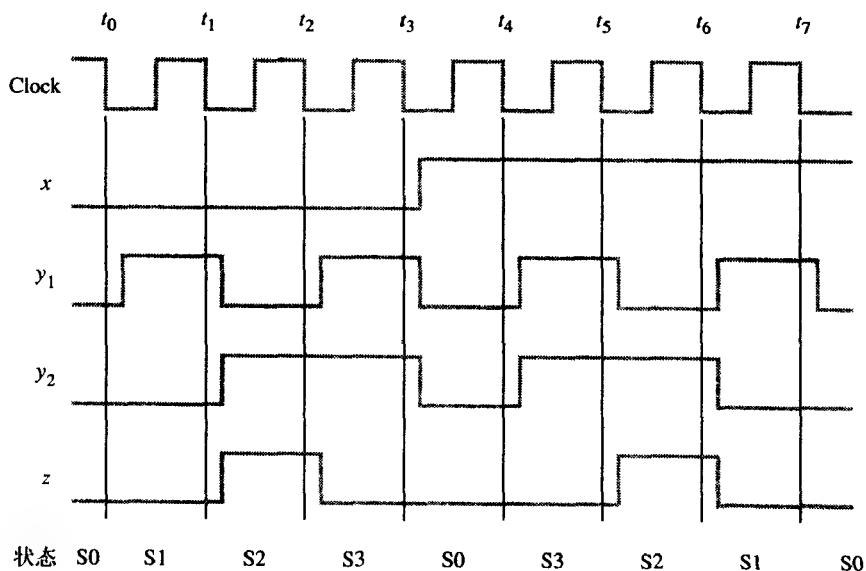


图A-50 升值/贬值计数器的实现

717

## A.13.2 时序图

了解计数器的时序图有助于完全掌握计数器电路的操作。图A-51给出了一个可能事件序列的例子。假设状态转换（改变触发器的值）发生在下降沿，并且计数器从状态S0开始。因为 $x=0$ ，计数器在 $t_0$ 时刻变为状态S1， $t_1$ 时刻变为S2， $t_2$ 时刻变为S3。当计数器进入状态S2时，输出从0变到1。当达到状态S3时又变到0。在S3状态末尾的 $t_3$ 时刻，计数器返回S0状态。假设在这一时刻，输入 $x$ 变为1，导致计数器贬值计数。当计数再一次达到S2时，即在 $t_5$ 时刻，输出 $z$ 变为1。



图A-51 图A-50所示电路的时序图

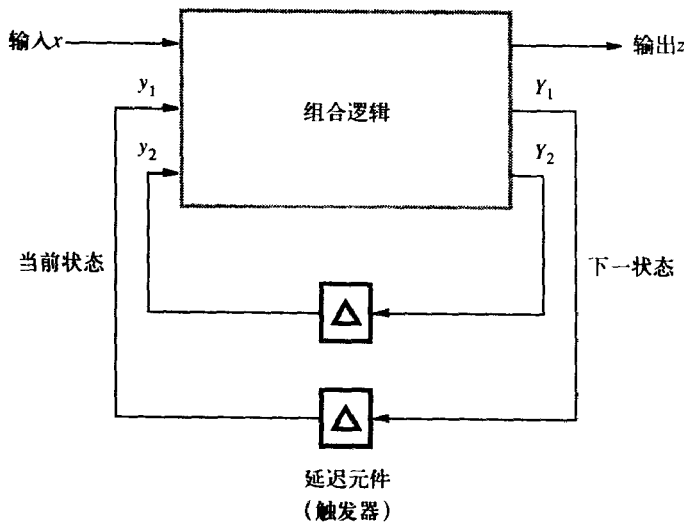
718

注意所有信号改变都在时钟下降沿之后立刻发生，并在下一个下降沿到来之前不会发生改变。从时钟边沿到变量 $y_i$ 改变之间的延迟是实现计数器电路触发器的传输延迟。还要注意我们假设输入 $x$ 也由同一个时钟控制，并且它的转变只发生在临近时钟周期开始的时刻。这是所有改变均由一个时钟控制的电路的基本性质。这样的电路称作同步时序电路。

另一个关注的重点是图A-47中状态图使用的标记与时序图的关系。例如，考虑 $t_1$ 与 $t_2$ 之间的时钟周期。在这个周期内，机器处于S2状态且输入值 $x = 0$ 。这一情况在状态图中用从S2状态发出标号为 $x = 0$ 的箭头表示。因为这个箭头指向S3状态，时序图显示在下一时钟边沿 $t_2$ ， $y_2$ 和 $y_1$ 变为S3状态相应的值。与箭头对应的输出值在S2状态赋值给 $z$ 。

### A.13.3 有限状态机模型

图A-50中使用触发器和组合逻辑门的同步时序电路实现升值/降值计数器的具体例子很容易概括为图A-52中的标准有限状态机模型。在这个模型中，延迟元件的时间延迟等于时钟周期的长度。这就是 $Y_i$ 发生变化到相应的 $y_i$ 发生变化的时间。模型假设组合逻辑块没有延迟；因此，输出 $z$ 、 $Y_1$ 和 $Y_2$ 是输入 $x$ 、 $y_1$ 和 $y_2$ 的瞬态函数。实际电路中的电路元件会产生一些延迟，如图A-51所示。如果组合逻辑块的延迟与时钟周期相比很小，电路就能正常工作。下一状态的输出 $Y_i$ 必须及时使触发器在时钟周期末尾改变到下一所需状态。另外，虽然输出 $z$ 可能在整个时钟周期内都没达到要求值，它必须在这个周期结束之前达到该值。



图A-52 有限状态机的标准模型

719

组合逻辑块的输入由表示当前状态的触发器输出 $y_i$ 和外部输入 $x$ 组成。块的输出是触发器输入 $Y_i$ 和外部输出 $z$ 。当有效时钟边沿到来结束当前时钟周期时， $Y_i$ 线的值被写入触发器。它成为状态变量 $y_i$ 的下一组值。因为这些信号连接在组合块的输入端上，所以它们和外部输入 $x$ 将产生新的 $z$ 和 $Y_i$ 值。一个时钟周期过去之后，新的 $Y_i$ 值被传送给 $y_i$ ，如此反复。换句话说，触发器形成了组合块从输出到输入的反饋回路，并引入了一个时钟周期的延迟。

尽管图A-52只显示了一个外部输入，一个外部输出和两个状态变量，但是显而易见，这三种变量的数目都是可以变化的。

### A.13.4 有限状态机的组合

让我们总结一下如何根据图A-47中的状态图设计具有图A-52基本结构的同步时序电路。设计的步骤如下:

1. 列出适当的状态图和状态表。
2. 确定所需触发器的数目, 选择合适类型的触发器。
3. 确定状态图中的每个状态需要在触发器中存储的值。这称为状态分配。
4. 列出状态赋值表。
5. 列出组合逻辑块的真值表。
6. 寻找实现组合逻辑块的适当电路。

#### 实例

以自动售货机作为兼具输入和输出的有限状态机的例子。为了简单起见, 假设机器只接受25美分的硬币和10美分的硬币。机器一直接受这两种硬币直至总金额达到30美分以上。当达到这个数目时, 机器提供一个输出(商品)。即使金额超过30美分也不找零。用二进制输入 $x_1$ 和 $x_2$ 来代表投入的硬币,  $x_1 = 1$ 或 $x_2 = 1$ 分别代表投入了一个25美分或一个10美分硬币。除此之外, 输入均为0。每次只能投入一枚硬币, 因此输入组合 $x_1 x_2 = 11$ 不可能发生。同样, 用二进制输出 $z$ 表示机器提供的商品, 即没有商品输出时 $z = 0$ , 而有商品输出时 $z = 1$ 。

设计售货机电路的第一步是画出状态图或状态表。最好能给出每个所需状态的口头说明, 然后再决定表示这些状态需要多少触发器。状态代表投币过程中任意时刻的总金额。由于可以按任何顺序投掷25美分硬币和10美分硬币直至总数等于或超过30美分, 因此所需的状态为:

$S_0$  = 未投币 (“起始”状态)

$S_1$  = 10美分

$S_2$  = 20美分

$S_3$  = 25美分

我们不需要更多的状态, 因为如果当前的状态是 $S_2$ 或 $S_3$ , 一个10美分硬币或一个25美分硬币都会补足当前输入, 产生输出 $z = 1$ 并将状态返回到 $S_0$ 重新开始。

图A-53给出了描述自动售货机电路所需行为的状态图。注意输入 $x_1 x_2 = 11$ 的情况不会出现, 因为不可能同时投掷两枚硬币。还要注意每个状态都有一个标记为00/0的箭头回指自己。这表明如果在一个时钟周期内没有硬币投入, 电路会保持它的当前状态。

这部机器只需要4个状态。这需要两个触发器。我们将它们标注为 $y_2$ 和 $y_1$ , 并且为各状态分配的值为 $S_0 = 00$ ,  $S_1 = 01$ ,  $S_2 = 10$ ,  $S_3 = 11$ , 图A-54给出了最终的状态分配表。在表中使用短线表示 $x_1 x_2 = 11$ 的输入组合不会出现。这些条目是无关项, 在设计组合逻辑块时可以利用, 稍后将作讨论。

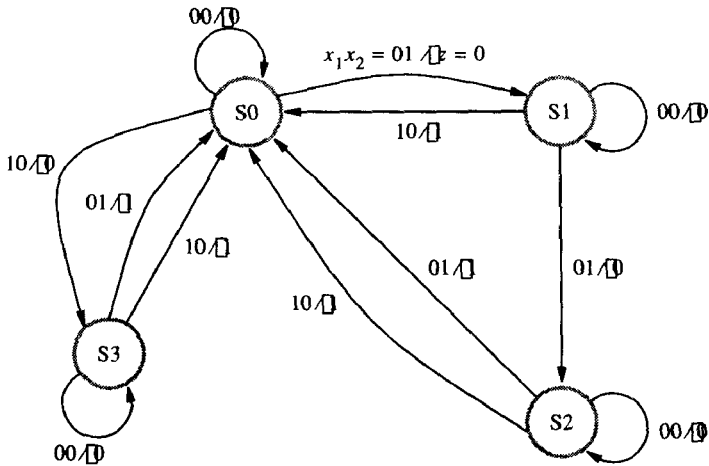
这就完成了组合过程的前4个步骤。现在我们进行第5步。图A-54中的状态分配表可以直接得出图A-55中描述组合逻辑块功能的真值表。从该表可以很容易地得出下面实现逻辑块的表达式:

$$Y_2 = \bar{x}_1 \bar{x}_2 y_2 + x_2 \bar{y}_2 y_1 + x_1 \bar{y}_2 \bar{y}_1$$

$$Y_1 = \bar{x}_1 \bar{x}_2 y_1 + \bar{y}_2 \bar{y}_1 (x_1 + x_2)$$

$$z = y_2 (x_1 + x_2) + x_1 y_1$$

注意到逻辑项 $\bar{x}_1 \bar{x}_2$ ,  $\bar{y}_1 \bar{y}_2$ 和 $(x_1 + x_2)$ 在不止一个表达式中出现。这在块的实现中会降低成本。



$x_1 = 1$  ~ 投入25美分硬币  
 $x_2 = 1$  ~ 投入10美分硬币  
 $z = 1$  ~ 给出商品  
          (共投入了30美分硬币)  
输入组合  $x_1x_2 = 11$  不可能发生

图A-53 自动售货机例子的状态表

当前 状态	下一状态				输出z			
	$x_1x_2 = 00$	$x_1x_2 = 01$	$x_1x_2 = 10$	$x_1x_2 = 11$	$x_1x_2 = 00$	$x_1x_2 = 01$	$x_1x_2 = 10$	$x_1x_2 = 11$
$y_2y_1$	$Y_2Y_1$	$Y_2Y_1$	$Y_2Y_1$	$Y_2Y_1$				
S0	0 0	0 1	1 1	-	0	0	0	-
S1	0 1	0 1	0 0	-	0	0	1	-
S2	1 0	1 0	0 0	-	0	1	1	-
S3	1 1	1 1	0 0	-	0	1	1	-

图A-54 自动售货机例子的状态分配表

时序电路可以很容易地用PAL、CPLD和FPGA实现，因为这些器件既包括触发器又包括组合逻辑门。现代的计算机辅助设计工具可以直接根据状态图的描述集成时序电路。

注意图A-54中的下一状态和输出项对S2和S3在状态改变发生时的所有输入组合都相同。这意味着并不真正需要两个状态来表示总金额为20美分和25美分的情况。只要一个状态就够了，因为无论当前金额是其中的哪一个，只要再投一枚硬币就会产生商品输出 ( $z = 1$ )，并使机器返回到开始状态S0。因此，状态S2和S3是等价的，并可由一个状态代替。这意味着实现机器只需三个状态，但仍然需要两个触发器。而在更一般的情况下，由状态等价而引起的状态数目的减少往往会使触发器数目减少，电路也会更简单。

实现时序电路时，在所需的组合逻辑中可以进一步节省资源。不同的状态分配会导致不同



的逻辑描述, 其中有些逻辑描述需要的门可能比其他的要少。我们不再深入研究这种思想, 但读者应当明白在合理设计和实现时序电路的过程中, 有许多有趣的问题需要考虑。

最后, 应当知道状态变量也可以用其他类型的触发器表示。为了使表示尽量简单, 在这里使用了D触发器。使用其他更灵活的触发器, 如JK触发器, 则所需的组合逻辑有时会减少。可参习题A.35和A.36对这一可能性的探讨。

前面所介绍的时序电路都是在同一时钟的控制下工作。不使用时钟也可以实现时序电路。这样的电路称为异步时序电路。它们的设计不像同步时序电路这样直接。为了对两种时序电路有完整的了解, 可以参考专门介绍逻辑设计的书籍<sup>[1, 3, 7-11]</sup>。

## A.14 结束语

723

本附录的主要目的是使读者了解逻辑设计的基本概念和计算机体系结构中普遍使用的电路结构。熟悉了这些知识就可以更好地掌握在本书主要章节中介绍的体系结构概念。正如我们多次提到的, 逻辑网络的详细设计需要借助CAD工具。这些工具考虑了许多细节问题, 可以被经验丰富的设计人员有效地利用。

IC技术和CAD工具的使用彻底改革了逻辑设计。市场上各种IC组件的成本不断降低, 而且新的发现和技术的发展一直在进行。这个附录介绍了在数字系统设计中有用的一些基本组件。

从设计者的角度来看, 最终电路的成本和速度是重要的参数。IC封装使用的数量越少, 能提高这些参数。使用大型的芯片, 在独立的芯片上实现复杂的逻辑网络, 就能达到这一目的。实际上, CPLD和FPGA器件在许多应用中都提供了有效的解决方案。

另外的两个设计目标变得越来越重要。结果电路的易测试性使得证明新设备运行正常和当出现问题时的修复工作变得更加容易。此外, 经常需要使用附加的、冗余的逻辑电路(例如, 复制一些部分)来提高系统的可靠性。这些可能会提高组件的成本。设计者的工作就是在这些目标中找到一个适当的平衡点。许多书籍<sup>[1, 12-17]</sup>讨论了测试和容错性问题。

## 习题

A.1 使用积之和形式实现COINCIDENCE函数, 其中COINCIDENCE =  $\overline{\text{XOR}}$ 。

724

A.2 使用数学公式和真值表证明下列等式:

(a)  $\overline{a \oplus b \oplus c} = \overline{abc} + \overline{abc} + \overline{abc} + \overline{abc}$

(b)  $x + w\overline{x} = x + w$

(c)  $x_1\overline{x}_2 + \overline{x}_2x_3 + x_3\overline{x}_1 = x_1\overline{x}_2 + x_3\overline{x}_1$

A.3 图PA-1给出了4个3变量函数 $f_1, f_2, f_3, f_4$ 的最小积之和形式。这些函数还有其他形式的最小表达式吗? 如果有, 将它们都写出来。

$x_1$	$x_2$	$y_2$	$y_1$	$Y_2$	$Y_1$	$z$
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	1	1	0
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	1	0
1	0	0	1	0	0	1
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	d	d	d
1	1	0	1	d	d	d
1	1	1	0	d	d	d
1	1	1	1	d	d	d

图A-55 自动售货机电路的  
组合逻辑描述

$x_1$	$x_2$	$x_3$	$f_1$	$f_2$	$f_3$	$f_4$
0	0	0	1	1	d	0
0	0	1	1	1	1	1
0	1	0	0	1	0	1
0	1	1	0	1	1	d
1	0	0	1	0	d	d
1	0	1	0	0	0	d
1	1	0	1	0	1	1
1	1	1	1	1	1	0

图PA-1 习题A.3的逻辑函数

A.4 利用无关项 $d$ 找出函数 $f$ 的最简积之和形式

$$f = x_1(x_2\bar{x}_3 + x_2x_3 + \bar{x}_2\bar{x}_3x_4) + x_2\bar{x}_4(\bar{x}_3 + x_1)$$

$$d = x_1\bar{x}_2(x_3x_4 + \bar{x}_3\bar{x}_4) + \bar{x}_1\bar{x}_3x_4$$

A.5 考虑下面的函数

$$f(x_1, \dots, x_4) = (x_1 \oplus x_3) + (x_1x_3 + \bar{x}_1\bar{x}_3)x_4 + x_1\bar{x}_2$$

(a) 使用卡诺图找到 $f$ 的最小成本积之和(SOP)表达式。

(b) 找到 $f$ 的互补项 $\bar{f}$ 的SOP表达式, 然后(使用德摩根律)对这个SOP表达式取反, 找到 $f$ 的表达式。得到的结果表达式应该是和之积形式(POS)。将它与(a)中所得的SOP表达式的成本进行比较。你可以从中得到什么结论?

A.6 写出函数 $f(x_1, x_2, x_3, x_4)$ 的最小成本实现, 其中如果一个或两个输入逻辑变量是1时,  $f = 1$ , 否则 $f = 0$ 。

725

A.7 图A-6定义了一个4位BCD码数字。设计一个电路, 它有4个输入, 标记为 $b_3, \dots, b_0$ 和一个输出 $f$ , 其中4位输入模式是有效BCD数时,  $f = 1$ ; 否则 $f = 0$ 。给出这个电路的最小成本实现。

A.8 两个2位数 $A = a_1a_0$ 和 $B = b_1b_0$ 要由4变量函数 $f(a_1, a_0, b_1, b_0)$ 进行比较。当满足

$$v(A) < v(B)$$

时, 函数的值为1。其中对任意2位数,  $v(X) = x_1 \times 2^1 + x_0 \times 2^0$ 。假设变量 $A$ 和 $B$ 满足 $|v(A) - v(B)| < 2$ 。以最少的门实现 $f$ 。

A.9 重复习题A.8, 要求当 $v(A) > v(B)$ 时, 函数 $f = 1$ ; 其输入满足 $v(A) + v(B) < 4$ 。

A.10 证明结合率对与非操作符不适用。

A.11 使用与非门(不超过6个)实现下面函数, 每个门有三个输入。假设原值和反值都可用。

$$f = x_1x_2 + x_1x_2x_3 + \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4$$

A.12 使用6个或更少的与非门实现下面函数。不能使用反输入变量。

$$f = x_1x_2 + \bar{x}_3 + \bar{x}_1x_4$$

A.13 只使用与非门, 尽可能经济地实现下面函数。不允许有反输入变量。

$$f = (x_1 + x_3)(\bar{x}_2 + \bar{x}_4)$$

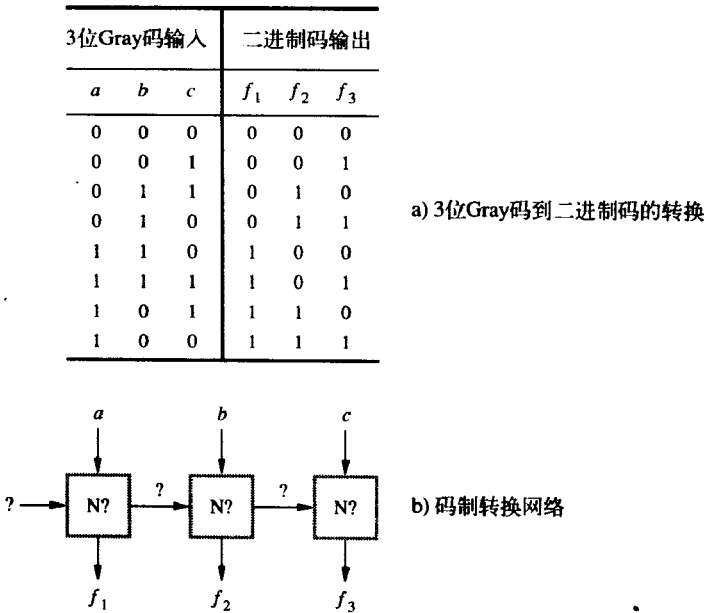
A.14 使用二进制表示的连续数字间只有1位不同的数字码称为Gray码。3位Gray码和二进制码变换的真值表如图PA-2a。

726

- (a) 只使用与非门实现函数 $f_1, f_2, f_3$ 。
- (b) 注意到下列输入和输出变量的关系，可以使这一码制转换的网络成本更低。

$$\begin{aligned}f_1 &= a \\f_2 &= f_1 \oplus b \\f_3 &= f_2 \oplus c\end{aligned}$$

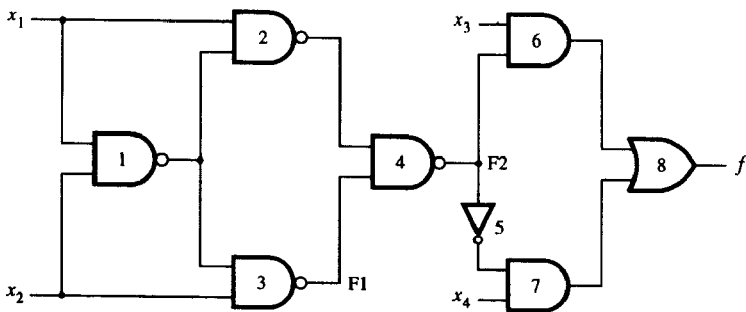
使用这些关系，指明可重复的组合网络N的内容，从而实现转换，如图PA-2b所示。比较这种形式下实现转换所需的与非门数量与a部分中的与非门总数。



图PA-2 习题A.14Gray码转换的例子

- A.15 使用4个2输入与非门实现异或函数。
- A.16 图A-37定义了一个BCD码七段显示器译码器。使用与、或、非门给出该真值表的实现。证明图中的与非门电路实现了同样的功能。
- A.17 在图PA-3的逻辑网络中，门3坏了，不管输入为何值，其输出F1都产生逻辑值1。重新画出网络图，尽可能化简，用最少的门得出一个与所给有故障网络等价的新网络。假设错误出在F2，它被所锁定在逻辑0，重复此问题。

727



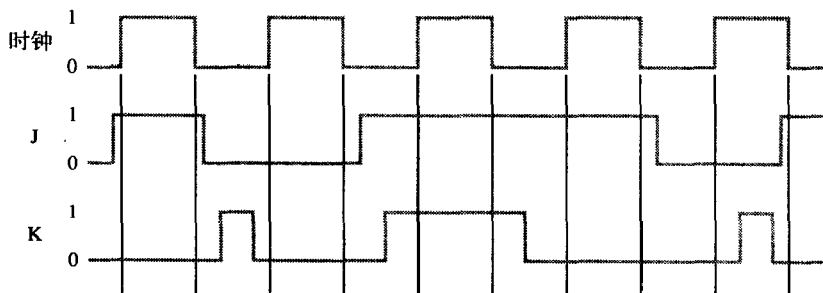
图PA-3 一个有故障的网络

A.18 图A-16显示了普通CMOS电路结构。设计一个CMOS电路实现下列函数。

$$f(x_1, \dots, x_4) = \bar{x}_1 \bar{x}_2 + \bar{x}_3 \bar{x}_4$$

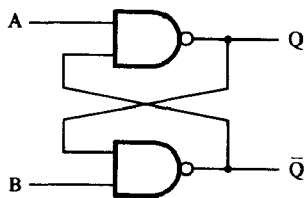
尽可能少用晶体管。(提示: 考虑晶体管的串联/并联网络。注意图A-17和图A-18中上拉和下拉网络相反的串联和并联结构。)

A.19 画出图A-31中JK电路输出Q的波形, 利用图PA-4的输入波形并假设触发器开始处于0状态。



图PA-4 JK触发器的输入波形

A.20 写出图PA-5中与或非门电路的真值表。将其与图A-24b中的真值表比较, 然后证明图A-26中的电路与图A-25a中的电路等价。



图PA-5 与非锁存器

- A.21 以或非门的延迟为单位, 计算图A-29中下降沿触发的D触发器的建立时间和保持时间。
- A.22 在图A-27a的电路中, 用或非门代替所有的与非门。写出结果电路的真值表。这个电路与图A-27a中的电路有何不同?
- A.23 图A-33显示了一个在时钟信号控制下每次将数据右移一位的移位寄存器网络。改进这个移位寄存器, 使它能够在时钟和附加控制输入ONE/TWO的共同作用下每次移动数据的一位或两位。
- A.24 我们需要一个4位移位寄存器, 它有两个控制输入端——INITIALIZE和 RIGHT/LEFT。当INITIALIZE置为1时, 二进制数1000被写入寄存器, 而与时钟输入无关。当INITIALIZE = 0时, 时钟输入的脉冲将这个模式循环移位。当RIGHT/LEFT的输入等于1或0时, 模式分别循环右移或循环左移。使用图A-32中具有预置和清除输入的D触发器, 给出这个寄存器的合理设计。
- A.25 构造一个3输入8输出的译码器网络, 规定所使用的门输入不得超过2个。
- A.26 图A-35显示了一个3位升值计数器。按相反顺序(即7, 6, ..., 1, 0, 7, ...)计数的计数器称为降值计数器。能在UP/DOWN信号控制下按两种顺序计数的计数器叫做升值/降值计数器。画出一个3位升值/降值计数器的逻辑图, 它能从外部源并行加载触发器, 从而可以被预置

为任何状态。LOAD/COUNT控制用来决定计数器是被加载还是做计数操作。

- A.27 图A-35显示了一个异步3位升值计数器。设计一个4位同步升值计数器，按0, 1, 2, ..., 15, 0, ...的顺序计数。在电路中使用T触发器。在同步计数器中，所有的触发器都能同时改变它们的状态。因此，主时钟输入应直接与所有触发器的时钟输入相连。

- A.28 实现如下表达式描述的开关函数

$$f(x_1, x_2, x_3, x_4) = x_1 x_3 \bar{x}_4 + \bar{x}_1 \bar{x}_3 x_4 + \bar{x}_2 \bar{x}_3 \bar{x}_4$$

(a) 使用8输入多路复用器电路实现 $f$ 。

(b) 可以用4输入多路复用器实现 $f$ 吗？如果可以，请写出方法。

- A.29 当

$$f(x_1, x_2, x_3, x_4) = x_1 \bar{x}_2 x_3 + x_2 x_3 x_4 + \bar{x}_1 \bar{x}_4$$

重复习题A.28。

- A.30 (a) 3个二进制定变量的函数 $f(x_1, x_2, x_3)$ ，其总数有多少？

(b) 这些函数中有多少可以用图A-43中的PAL电路实现？

(c) 如果任何3变量函数都能用一个PAL电路实现，那么对图A-43中电路做的最小改动是什么？

- A.31 考虑图A-43中的PAL电路。假设电路增加了第4个输入变量 $x_4$ ，其正向和反向形式可以和变量 $x_1, x_2, x_3$ 一样连接到所有4个与门。

(a) 这个修改过的PAL可以实现下面的函数吗？如果可以请写出方法。

$$f = x_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_3$$

(b) 多少个3变量的函数不能用这个PAL实现？

- A.32 使用状态分配 $S_0 = 10, S_1 = 11, S_2 = 01, S_3 = 00$ 完成对图A-47中升值/贬值计数器的设计。这个设计与A.13.1节的有什么不同？

- A.33 使用D触发器设计一个如图A-50中基本形式的2位同步计数器，按照...0, 3, 1, 2, 0, ...的顺序计数。这个电路没有外部输入，输出是触发器自己的值。

- A.34 重复习题A.33，设计一个按照...0, 1, 2, 3, 4, 5, 0, ...计数的三位计数器。设计组合逻辑时，将未使用的计数值6, 7作为无关项。

- A.35 A.13节使用D触发器设计同步时序电路。这是最简单的选择，因为D输入的逻辑函数值直接由状态表中所需的下一状态值决定。假设用JK触发器代替D触发器。建立一个表格，描述如何将触发器J和K输入端的二进制值确定为该触发器每个可能的从当前状态到下一状态迁移的函数。（提示：表应该有4行，每个迁移0→0, 0→1, 1→0, 1→1一行；并且每个J和K项是0, 1或无关项。）应用该表的信息为习题A.33中2位二进制计数器的两个触发器的J和K输入进行设计。这里所需的逻辑简化与使用D触发器进行的计数器设计有什么不同？

- A.36 使用JK触发器代替D触发器重复习题A.34。实现的步骤已经在习题A.35中给出。

- A.37 在A.13.4节说明有限状态机器模型的自动售货机例子中，使用了一个二进制输出 $z$ 表示货物的输出。找零不作为输出。本习题的目的是扩展输出，使其包括适当的找零。假设10美分和25美分的输入序列仅有10-10-10, 10-25, 25-10和25-25。与最后一个输入的硬币相对应，需要为这些序列提供的输出值分别是0, 5, 5和20。使用两个新的二进制输出 $z_2$ 和 $z_3$ 代表这三个不同的输出。（这并不直接与使用的硬币对应，但是问题会变得容易。）

(a) 描述包含新输出的状态表。

(b) 写出新输出 $z_2$ 和 $z_3$ 的逻辑表达式。

(c) 在新的状态表中存在等价状态吗?

A.38 有限状态机可以用来检测输入到机器的二进制序列中某些子序列的发生情况。这样的机器称为有限状态识别器。假设不论何时输入序列中出现模式011, 则机器在接收该模式的第二个1时在其输出产生1。

(a) 画出这台机器的状态图。

(b) 假设使用D触发器, 为所需数目的触发器进行状态分配并构造分配状态表。

(c) 写出输出变量和下一状态变量的逻辑表达式。

A.39 机器在输入序列中识别子序列011和010, 包括重叠出现的情况。重复习题A.38的a部分。例如, 当输入序列是110101011...时产生的输出序列是000010101...。

## 参考文献

1. S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, Burr Ridge, IL, 2000.
2. A.S. Sedra and K.C. Smith, *Microelectronic Circuits*, 4th ed., Oxford, New York, 1998.
3. J.F. Wakerley, *Digital Design Principles and Practices*, Prentice Hall, Upper Saddle River, NJ, 2000.
4. J.H. Jenkins, *Designing with FPGAs and CPLDs*, Prentice-Hall, Englewood Cliffs, N.J., 1994.
5. S.M. Trimberger, *Field-Programmable Gate Array Technology*, Kluwer, Boston, 1994.
6. S.D. Brown, R.J. Francis, J. Rose, and Z.G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer, Boston, 1992.
7. R.H. Katz, *Contemporary Logic Design*, Benjamin Cummings, Redwood City, Calif., 1994.
8. J.P. Hayes, *Digital Logic Design*, Addison-Wesley, Reading, Mass., 1993.
9. F.H. Hill and G.R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI*, 4th ed., Wiley, New York, 1993.
10. C.H. Roth, *Fundamentals of Logic Design*, 4th ed., West, St. Paul, Minn., 1992.
11. M.M. Mano and C.R. Kime, *Logic and Computer Design Fundamentals*, Prentice-Hall, Upper Saddle River, N.J., 1997.
12. M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, revised edition, IEEE Press, New York, 1995.
13. V.N. Yarmolik, *Fault Diagnosis of Digital Circuits*, Wiley, Chichester, England, 1994.
14. P.K. Lala, *Digital System Design Using Programmable Logic Devices*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
15. B.W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, Reading, Mass., 1989.
16. A.J. Miczo, *Digital Logic Testing and Simulation*, Wiley, New York, 1986.
17. D.K. Pradhan, *Fault-Tolerant Computing*, Prentice-Hall, Englewood Cliffs, N.J., 1986.

730

731  
732



## ARM指令集

我们在第3章第一部分中描述了v3版本ARM指令集体系 (ISA)，这里对其进行概要描述，同时对ISA的后续版本所作的改进进行简要的讨论。ARM寄存器结构如图3-1所示。指令集命令的一般格式在图3-2中给出。这里，我们将给出不同类型指令的详细内容。在ARM指令集中，所有的指令都被编码成32位的字。内存可以按字节寻址并且其地址长度是32位的。操作数大小有两种：字（32位）和字节（8位）。单字节操作数使用处理器寄存器的低8位。当一个单字节操作数被装入到一个寄存器时，其高位部分的三个字节都清成零。

## B.1 指令编码

图B-1给出了5种指令的编码情况。不同指令的类型是根据起始于 $b_{27}$ 的位模式来说明。图B-1b的乘法指令与B-1a中其他算术和逻辑指令组有所不同。在后一组中，当 $I = 0$ 时， $b_7$ 或 $b_4$ 位中有一个为0，但在乘法指令中这些位都是1。注意 $R_n$ 和 $R_d$ 字段在乘法指令中位置是颠倒的。

后面的几节给出了5种类型指令的编码细节以及相应的范例。完整的ARM体系结构具有与协处理器操作相关的附加指令。对此我们给出简要讨论。

733  
734

## 条件执行指令

表B-1中列出了条件执行指令的条件。所期望的条件以后缀的形式添加到指令OP（操作）码的助记符中。AL条件表明指令的执行不考虑条件码标志的状态。汇编语言程序中，默认状态是不写后缀的。例如，ADD（Add）和B（Branch）总是会被执行的，但是ADDEQ和BEQ只有 $Z = 1$ 的时候才能执行。条件指令经常与一个比较指令一同使用。在表B-1的名称一栏给出了它的使用方式。

## B.1.1 算术和逻辑指令

算术和逻辑指令与比较、检测和传送操作一样，也是采用图B-2中的指令格式。第一个操作数在寄存器 $R_n$ 中。第二个操作数或者是在寄存器 $R_m$ 中，或者是一个无符号的8位立即操作数（由第I位表明）。由4位OP码指定要进行的操作，操作的结果保存在寄存器 $R_d$ 中。如果S位等于1，说明条件码标志受结果影响，反之（ $S=0$ ）条件码标志不受结果的影响。

这一类指令的一般汇编语言形式是

OP{Cond}{S}  $R_d, R_n, \text{Operand 2}$

例如，如果第二个操作数存储在一个寄存器中（ $I=0$ ），指令



ADD R0,R1,R2

被无条件的执行, 进行操作

$R0 \leftarrow [R1] + [R2]$

而不影响条件码标志。如果OP码换成是ADDS, 那么操作的结果会影响到条件码标志。如果后续的指令是条件执行的, 假定条件是相等 (EQ), 则OP码将写作ADDEQS。

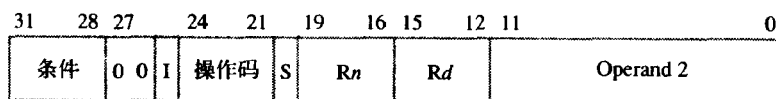
如果第二个操作数是一个立即值 (I=1), 用“#常数”表示。例如,

ADD R0, R1, #17

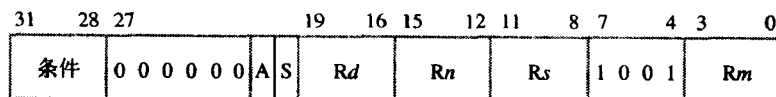
进行操作

$R0 \leftarrow [R1] + 17$

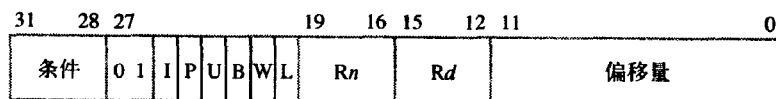
在使用该操作数之前, 立即数的值要用0扩展到32位。



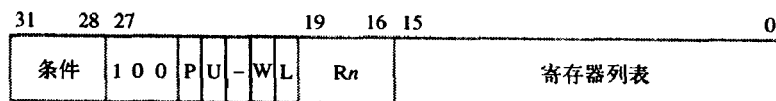
a) 算术、逻辑、比较、测试和传送指令



b) 乘法和乘法累加指令



c) 从/向存储器传送单字或字节指令



d) 从/向存储器传送多字指令



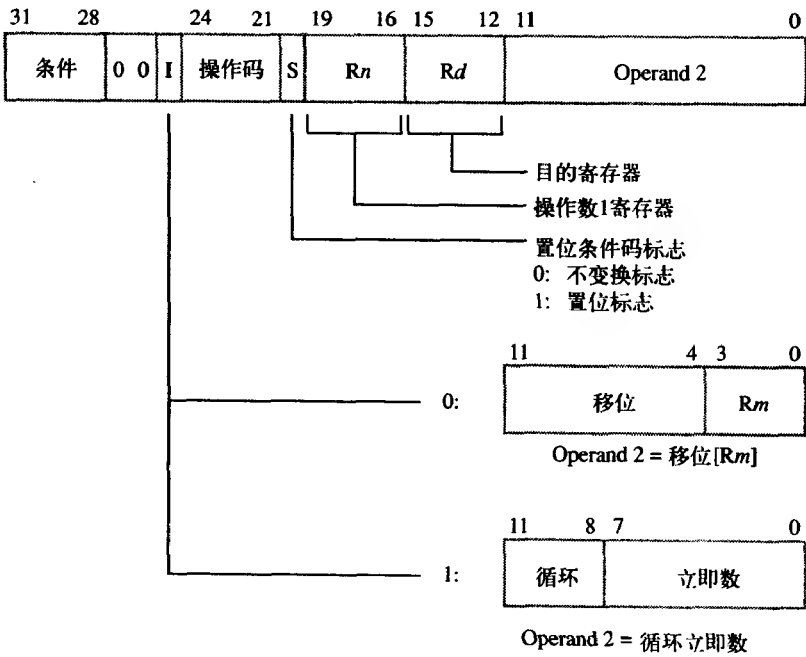
e) 传送指令和带有连接的传送指令

I	立即数	P	预/传递变址	W	写回
S	置位	U	升值/贬值	L	载入/存储
A	累加	B	字节/字	K	连接

图B-1 ARM指令编码格式

表B-1 ARM指令条件字段编码

条件字段 $b_{31} \cdots b_{28}$	条件后缀	名 称	条件码测试
0 0 0 0	EQ	相等 (0)	$Z = 1$
0 0 0 1	NE	不等 (非0)	$Z = 0$
0 0 1 0	CS/HS	进位置位/无符号大于或等于	$C = 1$
0 0 1 1	CC/LO	进位清除/无符号小于	$C = 0$
0 1 0 0	MI	减 (负数)	$N = 1$
0 1 0 1	PL	加 (正数或零)	$N = 0$
0 1 1 0	VS	溢出	$V = 1$
0 1 1 1	VC	不溢出	$V = 0$
1 0 0 0	HI	无符号大于	$\bar{C} \vee Z = 0$
1 0 0 1	LS	无符号小于或等于	$\bar{C} \vee Z = 1$
1 0 1 0	GE	有符号大于或等于	$N \oplus V = 0$
1 0 1 1	LT	有符号小于	$N \oplus V = 1$
1 1 0 0	GT	有符号大于	$Z \vee (N \oplus V) = 0$
1 1 0 1	LE	有符号小于或等于	$Z \vee (N \oplus V) = 1$
1 1 1 0	AL	一直有效	
1 1 1 1		不使用	



736

图B-2 ARM算术、逻辑、比较、测试和传送指令

第二个操作数的移位

• 如果第二个操作数保存在一个寄存器 ( $I = 0$ ) 中, 那么它可以在使用前进行移位, 如图B-3所示。移位体现在第 $b_{11-4}$ 位上。如果 $b_4 = 0$ , 在第 $b_{11-7}$ 位上用一个5位无符号数来表示一个从0到31的移动量。在 $b_6$ 和 $b_5$ 位中指明移位的类型。4种移位操作中所涉及的条件码标志C如图2-30和图2-32中所示。当移位的位数不为0时, 循环右移的操作 (ROR) 不需要C位 (进

位)。但是,当移位的位数是0时,则表示:带有(进位)C位的循环右移一位,如图2-32d所示。该操作可以使用汇编语言中的助记符RRX(扩展的循环右移)来表示,其中不指定移动位数。正如上面描述的那样,当在指令中直接指定移动的位数时,指令语法的形式可写成

ADD R0,R1,R2,LSL #4

执行的操作是将保存在R2的操作数向左移了4位(相当于乘以16),然后与R1中的内容相加。如果 $b_4 = 1$ ,移动的位数由寄存器Rs的低5位指明,如图B-3所示。例如,指令

ADD R0,R1,R2,LSR R3

将R2中内容向右移动,移动的位数由R3中的内容指定。

- 如果第二个操作数是一个立即数( $I = 1$ ),那么它将按照图B-2所示向右移。这选项通过对一个无符号的8位立即数进行循环右移来生成一个32位的大数。移动的位数是 $2n$ ,其中 $n$ 是 $b_{11-8}$ 中给出的4位数。因此,循环的范围是从0到30的偶数位。ARM指令集的这一特性从某种程度上补偿了指令中32位立即操作数的不足。但是,很明显,并非所有的值都能通过这种方法产生。

不过可以利用短序列指令,包括循环操作和OR 操作,来合成一个32位的数值,这个值是无法由一个单独的8位值经过循环操作而得到的。

表B-2和表B-3给出了完整的16位算术和逻辑指令集,以及将要在下一小节中讨论的两个乘法指令。该组指令中有6个关于加减的指令。其中进位加法和进位减法指令是用来对多字操作数进行操作的。因为只有第二个操作数可以被移位,所以提供了两个反向减法指令使得在减法操作中,被移位的操作数作为第一个操作数来使用。

在一个加法或减法指令中,如果一个立即值被当作第二操作数使用,那么它只能是一个正值。但是汇编语言允许使用指令

ADD R0,R1,# -5

和

SUB R0,R1,# -7

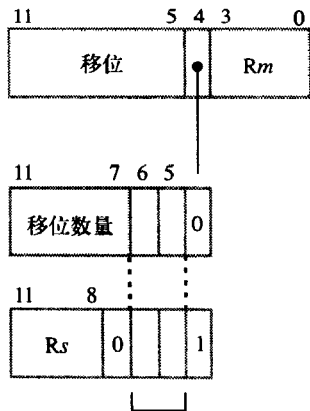
并把它们分别汇编成

SUB R0,R1,#5

和

ADD R0,R1,#7

除了4种逻辑指令,AND、ORR、EOR、BIC及传送补码指令(MVN)执行了逻辑非(NOT)操作。比较和测试指令通常会影影响条件码标志。



0 0	LSL	逻辑左移
0 1	LSR	逻辑右移
1 0	ASR	算术右移
1 1	ROR	循环右移

图B-3 对第二个操作数(图B-2)或寄存器Rm中的地址偏移量(图B-5)进行ARM移位操作

两条传送指令用来将第二操作数或它的按位补码传递给目的寄存器。第二操作数可以保存在寄存器中或者是一个立即操作数。因此，这两条指令除了可以进行寄存器传送操作以外，还可以将常数写入寄存器中。而MVN指令还可以装入一个用补码形式表示的负数。例如，指令

```
MOV R0, # -10
```

被汇编为

```
MVN R0, #9
```

因为 $9 = 0 \cdots 01001$ ，其补码是 $1 \cdots 10110$ ，即是-10的补码表示。

表B-2 ARM算术指令

助记符 (名称)	操作码	执行的操作	如果S=1受影响的CC标志			
	$b_{24} \cdots b_{21}$		N	Z	V	C
ADD (加)	0 1 0 0	$Rd \leftarrow [Rn] + Oper2$	x	x	x	x
ADC (带进位加)	0 1 0 1	$Rd \leftarrow [Rn] + Oper2 + [C]$	x	x	x	x
SUB (减)	0 0 1 0	$Rd \leftarrow [Rn] - Oper2$	x	x	x	x
SBC (带进位减)	0 1 1 0	$Rd \leftarrow [Rn] - Oper2 + [C] - 1$	x	x	x	x
RSB (反向减)	0 0 1 1	$Rd \leftarrow Oper2 - [Rn]$	x	x	x	x
RSC (带进位的反向减)	0 1 1 1	$Rd \leftarrow Oper2 - [Rn] + [C] - 1$	x	x	x	x
MUL (乘)	(参见图B-4)	$Rd \leftarrow [Rm] \times [Rs]$	x	x		
MLA (累加乘)	(参见图B-4)	$Rd \leftarrow [Rm] \times [Rs] + [Rn]$	x	x		

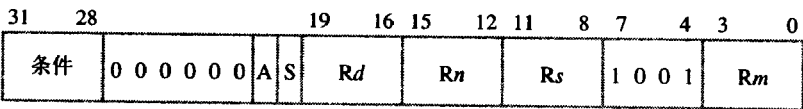
739

乘法指令

图B-4和表B-2分别给出了两条乘法指令的格式及操作。乘法指令中的任何一个操作数都不能进行移位操作。产生的乘积是一个单精度的32位值。

表B-3 ARM逻辑、比较、测试和传送指令

助记符 (名称)	操作码 $b_{24} \cdots b_{21}$	执行的操作	如果S = 1受影响的CC标志			
			N	Z	V	C
AND (逻辑与)	0 0 0 0	$Rd \leftarrow [Rn] \wedge Oper2$	x	x		x
ORR (逻辑或)	1 1 0 0	$Rd \leftarrow [Rn] \vee Oper2$	x	x		x
EOR (异或)	0 0 0 1	$Rd \leftarrow [Rn] \oplus Oper2$	x	x		x
BIC (位清除)	1 1 1 0	$Rd \leftarrow [Rn] \wedge \neg Oper2$	x	x		x
CMP (比较)	1 0 1 0	$[Rn] - Oper2$	x	x	x	x
CMN (比较负数)	1 0 1 1	$[Rn] + Oper2$	x	x	x	x
TST (位测试)	1 0 0 0	$[Rn] \wedge Oper2$	x	x		x
TEQ (测试相等)	1 0 0 1	$[Rn] \oplus Oper2$	x	x		x
MOV (传送)	1 1 0 1	$Rd \leftarrow Oper2$	x	x		x
MVN (补码传送)	1 1 1 1	$Rd \leftarrow \neg Oper2$	x	x		x



置位条件码标志  
0: 不改变标志  
1: 置位标志

累加标记  
0: 乘 (MUL)  
1: 累加乘 (MLA)

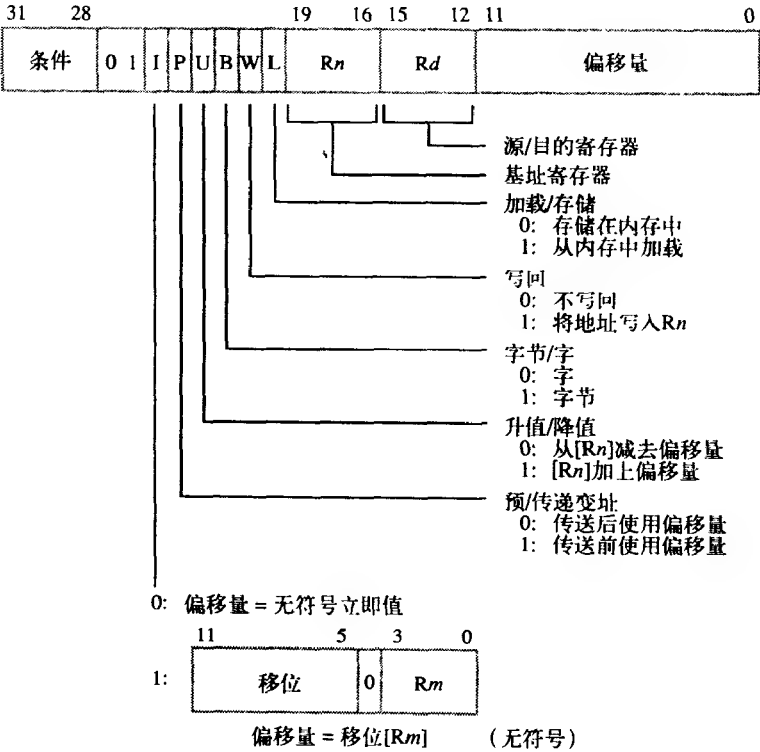
MUL:  $Rd \leftarrow [Rm] \times [Rs]$

MLA:  $Rd \leftarrow [Rm] \times [Rs] + [Rn]$

图B-4 ARM乘法和乘法累加指令

B.1.2 内存加载和存储指令

图B-5给出了访问内存的两条指令格式，表B-4则给出了它们相应的操作。在加载（LDR）指令中第L位（ $b_{20}$ ）是1而在存储（STR）指令中是0。在字节操作数中第B位（ $b_{22}$ ）是1而在32位字操作数中是0。一个字节的操作数通常保存在Rd的低位字节中。内存操作数的有效地址由寄存器Rn内容中的偏移量字段指定的偏移量来决定，是加（ $U = 1$ ）或减（ $U = 0$ ）。第P和W位决定了预变址还是传递变址以及写回操作，如图B-5和表3-1所示。注意对于寄存器Rm，只能使用这两种移位方法中的一种。



图B-5 ARM加载和存储指令

表B-4 向内存或从内存传送单字或字节的ARM指令

742

助记符(名称)	指令位	执行的操作
	B L	
LDR (加载字)	0 1	$Rd \leftarrow [EA]$
LDRB (加载字节)	1 1	$Rd \leftarrow [EA]$
STR (存储字)	0 0	$EA \leftarrow [Rd]$
STRB (存储字节)	1 0	$EA \leftarrow [Rd]$

下面是一个内存访问操作的例子。指令

LDR R0, [R1, #100]

执行的操作是

$R0 \leftarrow [[R1] + 100]$

它的位设置情况是I=0,P=1,U=1,B=0,W=0和L=1。偏移量的范围是 $\pm 4095$ 。指令

LDR R0, [R1, R2]

所执行的操作是

$R0 \leftarrow [[R1] + [R2]]$

第I位变为1，其他位的设置不变。

**743** 当偏移量被保存在寄存器中时，在与基址寄存器 $R_n$ 进行相加减之前，可以进行移位。这个移位操作只能使用图B-3所示的5位立即方式来指明。例如，指令

LDR R0, [R1, -R2, LSL #4]!

执行的操作是

$R0 \leftarrow [[R1] - 16 \times [R2]]$

并且有效地址被写回R1。这条指令的位设置情况是I=1, P=1, U=0, B=0, W=1和L=1。

如果程序计数器R15被指定为基址寄存器，则相应的寻址方式如表3-1所示。那么，产生操作数的有效地址时，使用的是带有一个立即数偏移量的不带写回的预变址方式。汇编语言允许对操作数的绝对地址进行命名。当指令执行时，汇编程序计算出与程序计数器的最新内容相对应的偏移量值。例如，如果指令

LDR R0, PARAMETER

被放在地址为1000的单元中，并且PARAMETER代表的地址为1100的单元，那么汇编程序就会生成指令

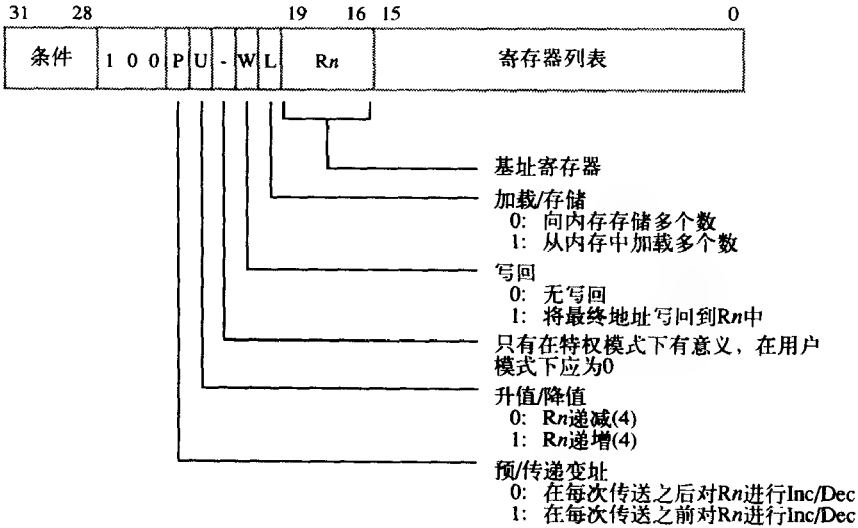
LDR R0, [R15, #92]

当偏移量被加到程序计数器的内容中时，计数器已经被更新为1008，因此偏移量必须是92才可以得到正确的访问地址 $1100=1008+92$ 。

### B.1.3 块加载与存储指令

图B-6 给出了在一块连续的内存字和16个处理器寄存器的特定子集之间传递数据的指令编码。OP码LDM用来将内存操作数加载到寄存器中，而STM 用来进行存储操作。在加载操作中 L 位是1而在存储操作中是0。通过16位寄存器列表的 $b_{15-0}$ 位中1的位置来指定所用到的寄存器。内存中字符块的开始位置由基址寄存器 $R_n$ 的内容指定。当第U位为1时块向高位地址运行，而当第U位为0时块向低位地址运行。由第P位指定 $R_n$ 采用的是预变址或传递变址。由于操作数通常是连续的4字节，所以变址值通常是4。当第W位为1时，执行成组传送所得的最终地址将要写回 $R_n$ ；反之（ $W = 0$ ）， $R_n$ 中保存开始的地址。不考虑是否块是向高位还是向低位地址执行，最小的寄存器经常与块中最低的地址值关联。表B-5给出了OP码对P、U和L位所有可能设置值的助记

符。操作码LDM和STM的后缀表明了第P位和第U位的设置值。例如，表B-5中的第一项 $P = 0$  和  $U = 1$ 由后缀IA表明，即“算后增加”，意思是基址寄存器 $Rn$ 的内容在每一次传送执行后加4，这表示 $Rn$ 是传递变址的。下面解释表B-5中列出的可以使用的助记符及其名称。



744

图B-6 ARM块传送指令

块传送的主要作用是对进入子程序和子程序返回的寄存器进行存储和恢复。如果我们假设用R13做堆栈指针，而R14（连接寄存器）保存返回地址，那么表B-5中最后一项的指令

```
STMDB R13!, {R0-R3, R14}
```

的功能是将寄存器R0中的内容通过R3和R14压入堆栈中。访问低位内存地址的堆栈和R0中的内容最后被传送到最低的地址中。表B-5中第一项给出了相应的指令

```
LDMIA R13!, {R0-R3, R15}
```

将R0到R3中保存的内容弹回先前那些寄存器，将R14中保存的值（返回地址）弹到程序计数器R15，完成返回操作。最后将最高地址的内容被传送到R15。这两个OP码的后缀DB和IA表示“先减”（DB）和“后增”（IA），描述了基址寄存器的内容是如何操作的。另一种相同功能的操作码助记符（参见表B-5）可以由STMFD和LDMFD的相同指令使用。后缀FD代表“满栈递减”，这是为了描述堆栈从低内存地址增长（减少）和基址寄存器R13中的初始内容是当前栈顶元素的地址（满栈）的情况。对于一个向高地址增长和使用堆栈指针越过当前顶元素指向空位置的堆栈来说，描述符的名称是“空栈递增”，后缀为EA。LDM和STM进入和返回中断服务程序指令的使用在第4章中已经讨论过。

745

B.1.4 转移与转移连接指令

图B-7给出了转移和转移连接指令的编码。偏移量是一个24位带符号数。它向左移动2位（所有的转移对象按字地址排列），符号扩展为32位，与当前的PC相加得到转移对象的地址。当前的PC指向转移指令下两个字（8字节）位置的指令。



表B-5 从或向内存传送多个字的ARM指令

助记符 (名称)	指令位			执行的操作
	P	U	L	
LDMIA/LDMFD 后增/满栈递减	0	1	1	$R_{low}, \dots, R_{high} \leftarrow [[R_n]], [[R_n] + 4], \dots$
LDMIB/LDMED 先增/空栈递减	1	1	1	$R_{low}, \dots, R_{high} \leftarrow [[R_n] + 4], [[R_n] + 8], \dots$
LDMDA/LDMFA 后减/满栈递增	0	0	1	$R_{high}, \dots, R_{low} \leftarrow [[R_n]], [[R_n] - 4], \dots$
LDMDB/LDMEA 先减/空栈递增	1	0	1	$R_{high}, \dots, R_{low} \leftarrow [[R_n] - 4], [[R_n] - 8], \dots$
STMIA/STMEA 后增/空栈递增	0	1	0	$[R_n], [R_n] + 4, \dots \leftarrow [R_{low}], \dots, [R_{high}]$
STMB/STMFA 先增/满栈递增	1	1	0	$[R_n] + 4, [R_n] + 8, \dots \leftarrow [R_{low}], \dots, [R_{high}]$
STMDA/STMED 后减/空栈递减	0	0	0	$[R_n], [R_n] - 4, \dots \leftarrow [R_{high}], \dots, [R_{low}]$
STMDB/STMFD 先减/满栈递减	1	0	0	$[R_n] - 4, [R_n] - 8, \dots \leftarrow [R_{high}], \dots, [R_{low}]$

746



K=0 : 转移(B)

K=1 : 转移连接(BL); 在寄存器R14中存储返回地址

图B-7 ARM转移与转移连接指令

在汇编语言中允许使用转移对象的绝对地址。例如，指令

BEQ ROUTINE

是一个条件转移指令，在条件Z=1成立时指向位置ROUTINE。如果转移指令地址是2000，ROUTINE地址是3000，汇编程序将计算出要插入指令的偏移量248。那么，实际的目标地址和当

前PC中的内容2008的实际距离,用程序计数器计算是992。这个值是248左移2位(相当于乘4)得来的。因此,转移地址的计算方法是 $2008+992=3000$ 。

转移连接指令(BL)被用来调用子程序。在转移到子程之前,紧跟在BL指令后面的指令的地址(返回地址)被保存在连接寄存器R14中。子程序的返回问题在3.6节中进行了介绍。

### B.1.5 机器控制指令

#### 软件中断

当执行完一个用户程序后,控制由一个软中断指令转移到超级用户程序。在第3章第一部分的例题程序中,我们没有给出SWI指令。在图3-8中程序的STR指令之后给出了需要立即使用该指令的一个例子。正如第4章描述的那样,SWI指令也是用来转换控制操作系统子程序中那些运行在超级用户模式下,为一个用户程序执行输入/输出的操作。

747

表B-6给出了汇编语言的SWI指令格式,其中包括执行的操作。操作码SWI在指令位字段的 $b_{27-24}$ 位,编码为1111。和其他ARM指令一样,它可以进行条件执行。指令低24位包含了一个在执行指令过程中被忽略的立即操作数。用户程序可以使用这个字段向操作系统传递一个参数来声明一个正在请求的服务,就像I/O操作一样。

#### 处理器状态寄存器传送

保存在当前处理器状态寄存器CPSR中的指令,和保存在处理器状态寄存器SPSR\_mode(见图3-1和图4-12)的指令主要被特权模式的程序所使用。在用户程序中这些指令也允许有限地被使用。当一个外部中断延迟了用户程序执行时,当前CPSR的内容自动保存在SPSR\_mode寄存器中,而调用一个特权模式子程序来处理中断。这些子程序必须熟练地处理状态寄存器中的内容。这些问题在第4章中进行了讨论。如表B-6所示,MRS和MSR指令用来对CPSR和SPSR\_mode寄存器进行读和写操作。用户模式和特权模式程序都可以读取状态寄存器。用户模式程序只可以改写CPSR寄存器的 $b_{31-28}$ 位的条件码标志N, Z, V, C。特权模式程序可以对CPSR和SPSR\_mode寄存器全部32位进行改写,还可以有选择地只对条件码标志进行改写。写操作的源操作数可以是一个通用寄存器中的内容,也可以是一个由表B-6中用imm32指明的一个的32位立即数值。立即操作数中只有高4位被使用,所以操作数通常可以由一条指令中的8位立即数通过循环移位得到。

在机器指令格式中对MRS和MSR的指令编码通常是用做算术和逻辑指令的(见图B-1)。如表B-3,这一组中的CMP、CMN、TST和TEQ指令经常用来置位条件码标志。因此,第S位在这些指令中常被置为1。当S位置为零,描述MRS和MSR指令的这四个操作码是对CPSR或SPSR\_mode寄存器进行操作。其他指令位用来区分MSR指令中的完全写入或部分写入,以及寄存器或立即源操作数。

#### 寄存器/内存对换

对于将内存中的内容读到一个寄存器中并且将另一个寄存器中的内容写入同一个内存单元中的不中断操作,系统中提供了一条指令。如表B-6所示的这个对换指令(助记符SWP),用户或特权模式都可以使用。它的主要作用是用执行锁存变量的操作,来调整多处理器结构间程序共享内存数据的正确运转。“不中断”意味由对换指令执行的读和写操作之间不允许另一个处理器访问内存。寄存器Rm和Rd可以是相同的,这样在寄存器和内存操作数之间实现了对换操作。

表B-6 状态寄存器传送、软件中断和数据对换的ARM指令

助记符(名称)	指令格式	执行的操作
MRS (复制状态寄存器)	用户模式:	
	MRS Rd,CPSR	$Rd \leftarrow [CPSR]$
	特权模式:	
	MRS Rd,CPSR	$Rd \leftarrow [CPSR]$
	MRS Rd,SPSR	$Rd \leftarrow [SPSR\_mode]$
MSR (写入状态寄存器)	用户模式:	
	MSR CPSR,Rm	$CPSR_{31-28} \leftarrow [Rm]_{31-28}$
	MSR CPSR,imm32	$CPSR_{31-28} \leftarrow imm32_{31-28}$
	特权模式:	
	MSR CPSR,Rm	$CPSR \leftarrow [Rm]$
	MSR CPSR_flg,Rm	$CPSR_{31-28} \leftarrow [Rm]_{31-28}$
	MSR CPSR_flg,imm32	$CPSR_{31-28} \leftarrow imm32_{31-28}$
	MSR SPSR,Rm	$SPSR\_mode \leftarrow [Rm]$
	MSR SPSR_flg,Rm	$SPSR\_mode_{31-28} \leftarrow [Rm]_{31-28}$
	MSR SPSR_flg,imm32	$SPSR\_mode_{31-28} \leftarrow imm32_{31-28}$
SWI (软件中断)	SWI imm24	$R14\_svc \leftarrow updated[PC];$ $SPSR\_svc \leftarrow [CPSR];$ $PC \leftarrow 0x08$
SWP (对换)	SWP Rd,Rm,[Rn]	$Rd \leftarrow [[Rn]];$ $[Rn] \leftarrow [Rm]$

748  
749

B.2 其他ARM指令

在这一节中，我们简要描述一下协处理器指令和在版本v4和v5体系结构中介绍的指令。

B.2.1 协处理器指令

不采用已定义的ARM指令集，而是通过硬部件来执行操作的硬件设备称作协处理器。其中浮点数操作的硬件设备就是一个例子。其他例子还有在嵌入式系统对数字信号或视频数据进行特殊处理的部件。如果在ARM处理器设计中提供了一个软件-综合形式，就像在第9章和第11章中描述的那样，那么就指定了一个协处理器的软件模块可以通过处理器的软件描述集成，并且得到一块单片机。在ARM指令集中包含的指令模板简化了组合单元的编程，它实现了指导协处理器执行操作，在协处理器的寄存器和内存间传递数据以及在协处理器的寄存器和ARM寄存

器间传递数据等功能。

### B.2.2 版本v4和v5指令

v3版之后的两个指令系统版本中包括了扩展的内存访问和乘法指令集。版本v4和v5具有加载和存储指令，它们在内存和处理器的寄存器间传递有符号字节和有符号/无符号的16位半字。在内部，ARM处理器只对32位操作数执行操作。当有符号字节和有符号半字使用v4和v5指令载入处理器的寄存器中时，它们要执行符号扩展成为32位。

v4和v5版本也提供了v3版（参见图B-4）中的MUL和MULA指令的附加形式，并给出了产生64位结果的这两条指令的有符号和无符号版本。

## B.3 编程实验

ARM网站（URL: [www.arm.com/hr.ns4/html/SDT202u](http://www.arm.com/hr.ns4/html/SDT202u)）包括了可以进入、编辑、汇编、运行（仿真）ARM汇编语言程序的软件开发工具。对于图3-8中的程序，AREA指示符应该改写成

```
AREA addloop, CODE
```

```
AREA addloopdata, DATA
```

来启动汇编程序。同样，正如在B.1.5节中所描述的，在SRT指令后需要一个形式为

```
SWI 0x123456
```

的软件中断指令。



## Motorola 68000 指令集

本附录包括了Motorola 68000 指令集的概要内容。在第3章的第二部分中我们给出了这款处理器主要特点的介绍性讨论，其中包括对寄存器结构和寻址方式的描述，这些内容分别包含在图3-18和表3-2中。注意表3-2还包括了寻址方式的汇编程序语法。

表C-1给出了为一个操作数地址字段编码的通用格式。它采用一个6位的字段指定寻址方式及使用的寄存器。对于那些无需指明特定寄存器的方式，全部的6位都用来指明寻址方式。

表C-1中寻址方式的命名与本书中使用的一样，其中有些与Motorola手册中的不同。因为读者会发现参考制造商数据一览表和用户手册是十分有帮助的，所以我们在表C-2中总结了两者使用术语上的区别。Motorola术语具有高度的描述性，但是在讨论时有些不太方便。

在本附录中以表格的形式列出68000指令。为了使该表格美观易读，表中使用了广义的符号缩写。表C-3给出了符号标志和它们的含义。注意在操作码字段中与位模式相应的符号每一位对应一个字母。

表C-1 68000的地址字段编码

地址字段					
方式			寄存器		
5	4	3	2	1	0
寻址方式			方式 字段	寄存器字段	
直接数据寄存器			000	寄存器号	
直接地址寄存器			001	寄存器号	
间接地址寄存器			010	寄存器号	
自动增量			011	寄存器号	
自动减量			100	寄存器号	
变址基址			101	寄存器号	
变址完全			110	寄存器号	
绝对短			111	000	
绝对长			111	001	
相对基址			111	010	
相对完全			111	011	
立即数或状态寄存器			111	100	

表C-4给出了变量指令的完整列表。每条指令允许的寻址方式都采用矩阵格式描述。对于每一个源（目标）操作数都提供了寻址方式，所有的目标（源）操作数允许的寻址方式由x表示。例如，对于AND指令来说，如果源是一个数据寄存器，则目标操作数的指定方式可能是 (An)、

(An)+、- (An)、d (An)、d (An, Xi)、Abs. W或者Abs.L。另外，如果目标是一个数据寄存器，源可以指定表中11种方式中的任意一种。

表C-2 与Motorola术语的不同

本书中使用的术语	Motorola中的术语
自增	地址寄存器间接后增
自减	地址寄存器间接先减
变址基址	地址寄存器用偏移量间接寻址
变址完全	地址寄存器用变址间接寻址
相对基址	带偏移量的程序计数器
相对完全	带有变址的程序计数器

表C-3 对表C-4中的注释

符号	含义
s	源操作数
d	目标操作数
An	地址寄存器 $n$
Dn	数据寄存器 $n$
Xn	用作变址寄存器的地址/数据寄存器
PC	程序计加器
SP	堆栈指针
SR	状态寄存器
CCR	SR中的条件码标志
AAA	地址寄存器数
DDD	数据寄存器数
rrr	源寄存器数
RRR	目标寄存器数
eeeeee	源操作数的有效地址
EEEEEE	目标操作数的有效地址
MMM	目标操作数的有效寻址方式
CCCC	指定条件码检测
P...P	位移量
Q...Q	快速立即数据
SS	长度: 00 = [byte, 01 = [word, 10 = [long word (对大多数指令) 01 = [byte, 11 = [word, 10 = [long word (对MOVE和MOVEA指令)
VVVV	陷阱向量数
u	条件码标志状态不定(无意义)
d(An)	变址基址寻址方式
d(An,Xi)	变址完全寻址方式
d(PC)	相对基址寻址方式
d(PC,Xi)	相对完全寻址方式

操作码一栏中给出了一条指令中前16位字的实际位模式。具有立即源数据的指令使用8位或16位操作数的第二个字，它的第2个和第3个字用于32位的操作数。对于变址寻址和相对寻址方式，所需要的变址值（即位移量）由操作码的下一个字给出。

移位和循环移位指令可以指明要被移位和循环移位操作数的位数。这个数量可以是数据寄存器的内容，也可以是包含在OP码中的3位立即数值。如果涉及一个内存操作数，这个数总是等于1。

表C-5中列出了转移指令。转移位移（偏移量）是一个指明字节相对距离的有符号的补码。对于条件转移指令和Scc（Set on condition）指令，表C-6给出了条件码后缀（cc）的可能情况。表中还给出了检测判断是否执行转移的条件。

表C-4和表C-5说明了对于一个给定的指令要执行的操作。对大多数指令来说，这个动作过程是显然的。但是，对于一部分指令来说，需要附加一些说明。助记符栏中由星号标记的指令将在接下来的几段中进行讨论。

#### BCHG、BCLR、BSET和BTST

所有这些指令都是对目标操作数中的一个特定位进行检测。需要测试位（bit#）的操作数可以是数据寄存器的内容，也可以是一个指令中的立即数。检测操作是将检测位的补码装入到条件标志Z中进行。

#### MOVEM

这条指令将一个或多个寄存器中的内容移进或移出一个连续的内存单元中。在传送中所涉及的寄存器由指令的第二个字指明。0到7位对应着D0到D7，8到15位对应着A0到A7。除了自动递减模式（在这种情况下寄存器的顺序是预先设定好的），这种方式对所有寻址方式都有效。

#### MOVEP

这个指令对68000与8位外围设备之间的数据传送很有用。数据是按字节传送的，内存地址在每个字节后增加2。这样，如果开始的地址是偶数，所有的字节根据数据总线的高8位线从偶数地址位置向内或向外传送。类似地，如果开始地址是奇数，所有的传送通过数据总线的低8位线来完成。数据寄存器的高位字节首先被传送，然后是低位字节。

68000有两个基本操作模式。在管态模式下，所有的指令都可以使用。在用户模式下，有些指令不能执行。只能在管态模式下使用的指令称为“特权指令”。它们是

- 当目标是状态寄存器SR时的ANDI、EORI、ORI和MOVE指令。
- 当从一个地址寄存器向内或向外传送用户堆栈指针时的MOVE指令。
- RESET、RTE和STOP指令。

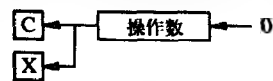
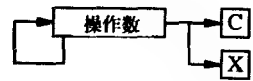
读者可以使用本附录中给出的信息来编写和调试68000的汇编语言程序。汇编指令的大小和结构是根据给出的OP码和使用的寻址方式来决定。由于篇幅有限，这里没有包含时序信息，比如需要执行一个给定指令的机器周期数等。这些信息以及指令集的更多具体细节可以在制造商手册中找到。



表C-4 68000

助记符 (名称)	长度	寻址方式	寻址方式												
			Dn	An	(An)	(An) +	(An) -	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR or CCR
ABCD (加BCD码)	B	s = Dn s = -(An)	d = x					x							
ADD (加)	B,W,L	s = Dn d = Dn	d = x s = x	x	x	x	x	x	x	x	x	x	x	x	x
ADDA (加地址)	W L	d = An d = An	s = x s = x	x	x	x	x	x	x	x	x	x	x	x	x
ADDI (加立即数)	B,W,L	s = Immed	d = x		x	x	x	x	x	x	x	x			
ADDQ (快速加)	B,W,L	s = Immed3	d = x	x	x	x	x	x	x	x	x	x			
ADDX (扩展加)	B,W,L	s = Dn s = -(An)	d = x					x							
AND (逻辑与)	B,W,L	s = Dn d = Dn	d = x s = x		x	x	x	x	x	x	x	x	x	x	x
ANDI (与立即数)	B,W,L	s = Immed	d = x		x	x	x	x	x	x	x	x			x
ASL (算术左移)	B,W,L	count = [Dn] count = QQQ count = 1	d = x d = x d =												
ASR (算术右移)	B,W,L	count = [Dn] count = QQQ count = 1	d = x d = x d =												
BCHG* (检测位并对其修改)	B L	bit# = [Dn] bit# = Immed bit# = [Dn] bit# = Immed	d = d = d = x d = x		x	x	x	x	x	x	x	x			
BCLR* (检测位并对其消除)	B L	bit# = [Dn] bit# = Immed bit# = [Dn] bit# = Immed	d = d = d = x d = x		x	x	x	x	x	x	x	x			

## 指令集

操作码 $b_{15} \dots b_0$	执行的操作	条件标志				
		X	N	Z	V	C
1100 RRR1 0000 0rrr 1100 RRR1 0000 1rrr	$d \leftarrow [s] + [d] + [X]$ 二进制编码的十进制加法	x	u	x	u	x
1101 DDD1 SSEE EEEE 1101 DDD0 SSee eeee	$d \leftarrow [Dn] + [d]$ $Dn \leftarrow [s] + [Dn]$	x	x	x	x	x
1101 AAA0 1lee eeee 1101 AAA1 1lee eeee	$An \leftarrow [s] + [An]$					
0000 0110 SSEE EEEE	$d \leftarrow s + [d]$	x	x	x	x	x
0101 QQQ0 SSEE EEEE	$d \leftarrow QQQ + [d]$	x	x	x	x	x
1101 RRR1 SS00 0rrr 1101 RRR1 SS00 1rrr	$d \leftarrow [s] + [d] + [X]$ 多精度加法	x	x	x	x	x
1100 DDD1 SSEE EEEE 1100 DDD0 SSee eeee	$d \leftarrow [Dn] \wedge [d]$		x	x	0	0
0000 0010 SSEE EEEE	$d \leftarrow s \wedge [d]$		x	x	0	0
1110 rrr1 SS10 0DDD 1110 QQQ1 SS00 0DDD 1110 0001 11EE EEEE		x	x	x	x	x
1110 rrr0 SS10 0DDD 1110 QQQ0 SS00 0DDD 1110 0000 11EE EEEE		x	x	x	x	x
0000 rrr1 01EE EEEE 0000 1000 01EE EEEE 0000 rrr1 01EE EEEE 0000 1000 01EE EEEE	$Z \leftarrow (\text{bit\# of } d);$ 之后对d中被检测位求补			x		
0000 rrr1 10EE EEEE 0000 1000 10EE EEEE 0000 rrr1 10EE EEEE 0000 1000 10EE EEEE	$Z \leftarrow (\text{bit\# of } d);$ 之后将d中的被检测位清除			x		

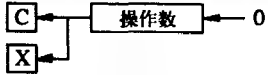
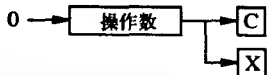
助记符 (名称)	长度	寻址方式	寻址方式													
			Dn	An	(An)	(An) +	-(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR or CCR	
BSET* (检测位年将其置位)	B	bit# = [Dn] d =			x	x	x	x	x	x	x					
	L	bit# = Immed d =			x	x	x	x	x	x	x					
		bit# = [Dn] d =	x													
		bit# = Immed d =	x													
BTST* (检测位)	B	bit# = [Dn] d =			x	x	x	x	x	x	x					
	L	bit# = Immed d =			x	x	x	x	x	x	x					
		bit# = [Dn] d =	x													
		bit# = Immed d =	x													
CHK (检查寄存器)	W	d = Dn	s =	x		x	x	x	x	x	x	x	x	x	x	
CLR (清零)	B,W,L		d =	x		x	x	x	x	x	x					
CMP (比较)	B,W,L	d = Dn	s =	x	x	x	x	x	x	x	x	x	x	x	x	
CMPA (比较地址)	W	d = An	s =	x	x	x	x	x	x	x	x	x	x	x	x	
	L	d = An	s =	x	x	x	x	x	x	x	x	x	x	x	x	
CMPI (比较立即数)	B,W,L	s = Immed	d =	x		x	x	x	x	x	x					
CMPM (比较内存)	B,W,L	s = (An)+	d =				x									
DIVS (有符号除法)	W	d = Dn	s =	x		x	x	x	x	x	x	x	x	x	x	
DIVU (无符号除法)	W	d = Dn	s =	x		x	x	x	x	x	x	x	x	x	x	
EOR (逻辑异或)	B,W,L	s = Dn	d =	x		x	x	x	x	x	x					
EORI (异或立即数)	B,W,L	s = Immed	d =	x		x	x	x	x	x	x				x	
EXG (交换)	L	s = Dn d =		x	x											
		s = An d =		x	x											
EXT (符号扩展)	W		d =	x												
	L		d =	x												

(续)

操作码 $b_{15} \dots b_0$	执行的操作	条件标志				
		X	N	Z	V	C
0000 rrr1 11EE EEEE 0000 1000 11EE EEEE 0000 rrr1 11EE EEEE 0000 1000 11EE EEEE	$Z \leftarrow (\text{bit\# of } d);$ 之后将d中检测的位置1			x		
0000 rrr1 00EE EEEE 0000 1000 00EE EEEE 0000 rrr1 00EE EEEE 0000 1000 00EE EEEE	$Z \leftarrow (\text{bit\# of } d);$			x		
0100 DDD1 10ee eeee	If $[Dn] < 0$ or $[Dn] > [s]$ , 之后产生一个中断		x	u	u	u
0100 0010 SSEE EEEE	$d \leftarrow 0$		0	1	0	0
1011 DDD0 SSee eeee	$[d] - [s]$		x	x	x	x
1011 AAA0 11ee eeee 1011 AAA1 11ee eeee	$[An] - [s]$		x	x	x	x
0000 1100 SSEE EEEE	$[d] - [s]$		x	x	x	x
1011 RRR1 SS00 lrrr	$[d] - [s]$		x	x	x	x
1000 DDD1 11ee eeee	$d \leftarrow [d] + [s]$ , using 其中d是32位, s为16位		x	x	x	0
1000 DDD0 11ee eeee	$d \leftarrow [d] + [s]$ , using 其中d是32位, s为16位		x	x	x	0
1011 rrr1 SSEE EEEE	$d \leftarrow [Dn] \oplus [d]$		x	x	0	0
0000 1010 SSEE EEEE	$d \leftarrow s \oplus [d]$		x	x	0	0
1100 DDD1 0100 0DDD 1100 AAA1 0100 1AAA 1100 DDD1 1000 1AAA	$[s] \leftrightarrow [d]$					
0100 1000 1000 0DDD 0100 1000 1100 0DDD	(bits 15-8 of d) $\leftarrow$ (bit 7 of d) (bits 31-16 of d) $\leftarrow$ (bit 15 of d)		x	x	0	0
			x	x	0	0

助记符 (名称)	长度	寻址方式	寻址方式											SR or CCR
			Dn	An	(An)	(An) + -(An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	
JMP (跳转)		d =			x		x	x	x	x	x	x		
JSR (跳转到子程序)		d =			x		x	x	x	x	x	x		
LEA (载入有效地址)	L	d = An	s =		x		x	x	x	x	x	x		
LINK (连接与分配)		disp = Immed	s =	x										
LSL (逻辑左移)	B,W,L	count = [Dn]	d =	x										
	W	count = QQQ	d =	x										
		count = I	d =		x	x	x	x	x	x				
LSR (逻辑右移)	B,W,L	count = [Dn]	d =	x										
	W	count = QQQ	d =	x										
		count = I	d =		x	x	x	x	x	x				
MOVE (传送)	B,W,L	s = Dn	d =	x	x	x	x	x	x	x				
		s = An	d =	x	x	x	x	x	x	x				
		s = (An)	d =	x	x	x	x	x	x	x				
		s = (An)+	d =	x	x	x	x	x	x	x				
		s = -(An)	d =	x	x	x	x	x	x	x				
		s = d(An)	d =	x	x	x	x	x	x	x				
		s = d(An,Xi)	d =	x	x	x	x	x	x	x				
		s = Abs.W	d =	x	x	x	x	x	x	x				
		s = Abs.L	d =	x	x	x	x	x	x	x				
		s = d(PC)	d =	x	x	x	x	x	x	x				
		s = d(PC,Xi)	d =	x	x	x	x	x	x	x				
		s = Immed	d =	x	x	x	x	x	x	x				
	W	d = CCR	s =	x	x	x	x	x	x	x	x	x	x	
		d = SR	s =	x	x	x	x	x	x	x	x	x	x	
	L	s = SP	d =	x	x	x	x	x	x	x				
		d = SP	s =	x	x									
MOVEA (传送地址)	W,L	d = An	s =	x	x	x	x	x	x	x	x	x	x	

(续)

操作码 $b_{15} \dots b_0$	执行的操作	条件标志				
		X	N	Z	V	C
0100 1110 11EE EEEE	$PC \leftarrow d$ 的有效地址					
0100 1110 10EE EEEE	$SP \leftarrow [SP] - 4;$ $[SP] \leftarrow [PC];$ $PC \leftarrow d$ 的有效地址					
0100 AAA1 11ee eeee	$An \leftarrow s$ 的有效地址					
0100 1110 0101 0AAA	$SP \leftarrow [SP] - 4;$ $[SP] \leftarrow [An];$ $An \leftarrow [SP];$ $SP \leftarrow [SP] + disp$					
1110 $m1$ SS10 1DDD 1110 QQQ1 SS00 1DDD 1110 0011 11EE EEEE		x	x	x	0	x
1110 $m0$ SS10 1DDD 1110 QQQ0 SS00 1DDD 1110 0010 11EE EEEE		x	x	x	0	x
00SS RRRM MMee eeee	$d \leftarrow [s]$		x	x	0	0
0100 0100 11ee eeee 0100 0110 11ee eeee 0100 0000 11EE EEEE 0100 1110 0110 1AAA 0100 1110 0110 0AAA	$CCR \leftarrow [\text{bits } 7-0 \text{ of } s]$ $SR \leftarrow [s]$ $d \leftarrow [SR]$ $d \leftarrow [SP]$ $SP \leftarrow [d]$	x x	x x	x x	x x	x x
00SS AAA0 01ee eeee	$An \leftarrow [s]$					

助记符 (名称)	长度	寻址方式	寻址方式											Immed	SR or CCR
			Dn	An	(An)	(An) +	(An) -	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)		
MOVEM* (传送多个寄存器)	W	s = Xn	d =		x		x	x	x	x	x				
		d = Xn	s =		x	x		x	x	x	x	x	x		
	L	s = Xn	d =		x		x	x	x	x	x				
		d = Xn	s =		x	x		x	x	x	x	x	x		
MOVEP* (传送外围数据)	W	s = Dn	d =						x						
	L	s = Dn	d =						x						
	W	s = d(An)	d =	x											
	L	s = d(An)	d =	x											
MOVEQ (快速传送)	L	s = Immed8	d =	x											
MULS (有符号乘)	W	d = Dn	s =	x	x	x	x	x	x	x	x	x	x	x	
MULU (无符号乘)	W	d = Dn	s =	x	x	x	x	x	x	x	x	x	x	x	
NBCD (非BCD)	B		d =	x	x	x	x	x	x	x	x				
NEG (非)	B,W,L		d =	x	x	x	x	x	x	x	x				
NEGX (不扩展)	B,W,L		d =	x	x	x	x	x	x	x	x				
NOP (无操作)															
NOT (求补)	B,W,L		d =	x	x	x	x	x	x	x	x				
OR (逻辑或)	B,W,L	s = Dn	d =		x	x	x	x	x	x	x				
		d = Dn	s =	x	x	x	x	x	x	x	x	x	x	x	
ORI (或立即数)	B,W,L	s = Immed	d =	x	x	x	x	x	x	x	x				x
PEA (有效地址压栈)	L		s =		x			x	x	x	x	x	x		

(续)

操作码 $b_{15} \dots b_0$	执行的操作	条件标志				
		X	N	Z	V	C
0100 1000 10EE EEEE 0100 1100 10ee eeee 0100 1000 11EE EEEE 0100 1100 11ee eeee	$d \leftarrow [Xn]$ $Xn \leftarrow [s]$ $d \leftarrow [Xn]$ $Xn \leftarrow [s]$	第二个字被用来 指明所涉及的寄 存器				
0000 DDD1 1000 1AAA 0000 DDD1 1100 1AAA 0000 DDD1 0000 1AAA 0000 DDD1 0100 1AAA	$d$ 的可用字节 $\leftarrow [Dn]$ $Dn \leftarrow d$ 的可用字节					
0111 DDD0 QQQQ QQQQ	$Dn \leftarrow QQQQQQQQ$		x	x	0	0
1100 DDD1 11ee eeee	$Dn \leftarrow [s] \times [Dn]$		x	x	0	0
1100 DDD0 11ee eeee	$Dn \leftarrow [s] \times [Dn]$		x	x	0	0
0100 1000 00EE EEEE	$d \leftarrow 0 - [d] - [X]$ 使用BCD算术	x	u	x	u	x
0100 0100 SSEE EEEE	$d \leftarrow 0 - [d]$	x	x	x	x	x
0100 0000 SSEE EEEE	$d \leftarrow 0 - [d] - [X]$	x	x	x	x	x
0100 1110 0111 0001	none					
0100 0110 SSEE EEEE	$d \leftarrow \overline{[d]}$		x	x	0	0
1000 DDD1 SSEE EEEE 1000 DDD0 SSee eeee	$d \leftarrow [s] \vee [d]$		x	x	0	0
0000 0000 SSEE EEEE	$d \leftarrow s \vee [d]$		x	x	0	0
0100 1000 01ee eeee	$SP \leftarrow [SP] - 4;$ $[SP] \leftarrow s$ 的有效地址					



				寻址方式											
助记符 (名称)	长度	寻址方式		Dn	An	(An)	(An) + - (An)	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)	Immed	SR or CCR
复位															
ROL	B,W,L	count = [Dn]	d =	x											
(不带X的循环左移)	W	count = QQQ	d =	x											
		count = 1	d =			x	x	x	x	x	x				
ROR	B,W,L	count = [Dn]	d =	x											
(不带X的循环右移)	W	count = QQQ	d =	x											
		count = 1	d =			x	x	x	x	x	x				
ROXL	B,W,L	count = [Dn]	d =	x											
(带X的循环左移)	W	count = QQQ	d =	x											
		count = 1	d =			x	x	x	x	x	x				
ROXR	B,W,L	count = [Dn]	d =	x											
(带X的循环右移)	W	count = QQQ	d =	x											
		count = 1	d =			x	x	x	x	x	x				
RTE															
(从异常返回)															
RTR															
(返回并恢复CCR)															
RTS															
(从子程序返回)															
SBCD	B	s = Dn	d =	x											
(减BCD)		s = - (An)	d =				x								
Sec	B		d =	x	x	x	x	x	x	x	x				
(设置条件)															
STOP			s =												x
(载入SR并停止)															

(续)

操作码 $b_{15} \dots b_0$	执行的操作	条件标志					
		X	N	Z	V	C	
0100 1110 0111 0000	断言RESET输出行						
1110 rrr1 SS11 1DDD 1110 QQQ1 SS01 1DDD 1110 0111 11EE EEEE			x	x	0	x	
1100 rrr1 SS11 1DDD 1110 QQQ0 SS01 1DDD 1110 0111 11EE EEEE			x	x	0	x	
1110 rrr1 SS11 0DDD 1110 QQQ1 SS01 0DDD 1110 0101 11EE EEEE		x	x	x	0	x	
1110 rrr0 SS11 0DDD 1110 QQQ0 SS01 0DDD 1110 0100 11EE EEEE		x	x	x	0	x	
0100 1110 0111 0011	SR ← [[SP]]; SP ← [SP] + 2; PC ← [[SP]]; SP ← [SP] + 4;	x	x	x	x	x	
0100 1110 0111 0111	CCR ← [[SP]]; SP ← [SP] + 2; PC ← [[SP]]; SP ← [SP] + 4;	x	x	x	x	x	
0100 1110 0111 0101	PC ← [[SP]]; SP ← [SP] + 4						
1000 RRR1 0000 0rrr 1000 RRR1 0000 1rrr	$d \leftarrow [d] - [s] - [X]$ 二进制编码的十进制减法	x	u	x	u	x	
0101 CCCC 11EE EEEE	如果cc是真, 将d的全部8位置成1, 否则, 将它们清成0						
0100 1110 0111 0010	SR ← s; 等待中断	x	x	x	x	x	

助记符 (名称)	长度	寻址方式	寻址方式											SR or CCR	
			Dn	An	(An)	(An) +	(An) -	d(An)	d(An,Xi)	Abs.W	Abs.L	d(PC)	d(PC,Xi)		Immed
SUB (减)	B,W,L	s = Dn d = Dn	d = x s = x	x	x	x	x	x	x	x	x	x	x	x	x
SUBA (减地址)	W L	d = An d = An	s = x s = x	x	x	x	x	x	x	x	x	x	x	x	x
SUBI (减立即数)	B,W,L	s = Immed	d = x		x	x	x	x	x	x	x	x			
SUBQ (快速减)	B,W,L	s = Immed3	d = x	x	x	x	x	x	x	x	x	x			
SUBX (扩展减法)	B,W,L	s = Dn s = -(An)	d = x d =					x							
SWAP (对换寄存器的高低位)	W		d = x												
TAS (检测与置位)	B		d = x		x	x	x	x	x	x	x	x			
TRAP (陷阱)		s = Immed4													
TRAPV (溢出陷阱)															
TST (检测)	B,W,L		d = x		x	x	x	x	x	x	x	x			
UNLK (非连接)				x											

(续)

操作码 $b_{15} \dots b_0$	执行的操作	条件标志				
		X	N	Z	V	C
1001 DDD1 SSEE EEEE 1001 DDD0 SSee eeee	$d \leftarrow [d] - [s]$	x	x	x	x	x
1001 AAA0 1lee eeee 1001 AAA1 1lee eeee	$An \leftarrow [An] - [s]$					
0000 0100 SSEE EEEE	$d \leftarrow [d] - s$	x	x	x	x	x
0101 QQQ1 SSEE EEEE	$d \leftarrow [d] - QQQ$	x	x	x	x	x
1001 RRR1 SS00 0rrr 1001 RRR1 SS00 1rrr	$d \leftarrow [d] - [s] - [X]$	x	x	x	x	x
0100 1000 0100 0DDD	$[Dn]_{31-16} \leftrightarrow [Dn]_{15-0}$		x	x	0	0
0100 1010 11EE EEEE	测试d并置位N与Z标志; 将d的第7位置1		x	x	0	0
0100 1110 0100 VVVV	$SP \leftarrow [SP] - 4;$ $[SP] \leftarrow [PC];$ $SP \leftarrow [SP] - 2;$ $[SP] \leftarrow [SR];$ $PC \leftarrow \text{vector}$					
0100 1110 0111 0110	If $V = 1$ , then $SP \leftarrow [SP] - 4;$ $[SP] \leftarrow [PC];$ $SP \leftarrow [SP] - 2;$ $[SP] \leftarrow [SR];$ $PC \leftarrow \text{TRAPV vector}$					
0100 1010 SSEE EEEE	测试d并置位N和Z标志		x	x	0	0
0100 1110 0101 1AAA	$SP \leftarrow [An];$ $An \leftarrow [[SP]];$ $SP \leftarrow [SP] + 4$					

表C-5 68000的转移指令

助记符 (名称)	偏移量长度	操作码	执行的操作
BRA	8	0100 0000 PPPP PPPP	$PC \leftarrow [PC] + disp$
(总是转移)	16	0110 0000 0000 0000 PPPP PPPP PPPP PPPP	
Bcc	8	0110 CCCC PPPP PPPP	If cc is true, then
(条件转移)	16	0110 CCCC 0000 0000 PPPP PPPP PPPP PPPP	$PC \leftarrow [PC] + disp$
BSR	8	0110 0001 PPPP PPPP	$SP \leftarrow [SP] - 4;$ $[SP] \leftarrow [PC];$
(转到子程序)	16	0110 0001 0000 0000 PPPP PPPP PPPP PPPP	$PC \leftarrow [PC] + disp$
DBcc	16	0101 CCCC 1100 1DDD PPPP PPPP PPPP PPPP	If cc is false, then $Dn \leftarrow [Dn] - 1;$ If $[Dn] \neq -1$ , then $PC \leftarrow [PC] + disp$
DBRA	汇编程序将此指令翻译为DBF (参见DBcc表项)		
(递减并转移)			

766  
768

表C-6 用于Bcc、DBcc和Scc指令的条件码

机器码 CCCC	条件后缀	名称	检测条件
0000	T	真	总为真
0001	F	假	总为假
0010	HI	高	$C \vee Z = 0$
0011	LS	小于或相等	$C \vee Z = 1$
0100	CC	进位清零	$C = 0$
0101	CS	进位置位	$C = 1$
0110	NE	不等	$Z = 0$
0111	EQ	相等	$Z = 1$
1000	VC	溢出清零	$V = 0$
1001	VS	溢出置位	$V = 1$
1010	PL	加	$N = 0$
1011	MI	减	$N = 1$
1100	GE	大于或等于	$N \oplus V = 0$
1101	LT	小于	$N \oplus V = 1$
1110	GT	大于	$Z \vee (N \oplus V) = 0$
1111	LE	小于或等于	$Z \vee (N \oplus V) = 1$

注：T和F后缀不能用在Bcc指令中。

Intel IA-32指令集

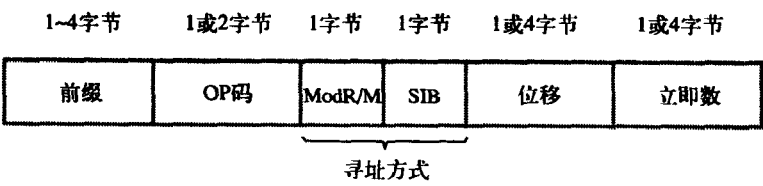
本附录是对第3章第三部分介绍的Intel IA-32指令集的概述。这个指令集十分庞大。我们只描述其中很小的一部分，大约50条指令，其中包括了第3章中所有用到的指令。另外还概要介绍了其他指令类型的一些特点。

图3-37和图3-38给出了IA-32寄存器的结构，我们在3.16.1节中对其做了描述。图3-41 给出了指令的一般格式。内存是按字节寻址的，且地址长度是32位的。指令中可以使用两种操作数：双字（32位）和字节（8位）。早期的16位Intel处理器使用字操作数（16位）。IA-32处理器可以按16位模式执行早期16位处理器所编写的机器程序。

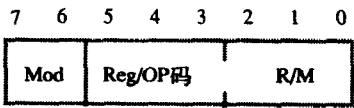
D.1 指令编码

图D-1a给出了IA-32指令编码的一般格式。OP码长度可以是1或2个字节。如图D-1b所示，对于一些指令来说，OP码在ModR/M字节被扩展到3位Reg/OP码字段。由于OP码的编码非常不规则，所以不再进行详细的讨论。前缀字节是大多数普通指令编码时所不需要的，我们将在D.3节中进行介绍。

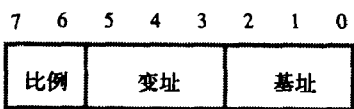
769  
1  
770



a) 一般格式



b) ModR/M字节



c) SIB字节

图D-1 IA-32指令格式

一条指令长度的范围可以从1个字节（一个OP码）到11或更多字节（即当同时指明了一个4字节的位移量和一个4字节的立即操作数以及2个寻址方式字节和OP码时）。例如，对一个寄存器操作数的递增（INC）和递减（DEC）指令只需要一个3位字段的单字节OP码来命名寄存器。下面举一个长指令的例子：

```
MOV DWORD PTR [EBP + ESI*4 + DISP], 10
```

正如在第3.17.1节中讨论过的那样，这个指令编码需要11个字节。在那一节中还举了一些其他指令的例子。

操作数的数据大小（8位或32位）在OP码中指出。OP码还指出在指令中是否有一个操作数是立即数，如果有的话，该立即数包含在指令中的最后一个字段中。

一个双操作数指令中至少有一个操作数是必须放在寄存器中的。它是在ModR/M字节的Reg/OP字段中指定的。寄存器的3位码如表D-1所示。如果另一个操作数也在寄存器中，它会在同一个字节的R/M字段中命名。如果另一个操作数不在寄存器里，它可能是一个立即值或者是一个内存单元的内容。下一节中将介绍，一个内存操作数的地址由两个寻址方式字节和偏移量指明。正如3.17.1节中所描述的，在对一个双操作数指令编码的过程中，哪个操作数是源由OP码中的方向位来决定。

表D-1 IA-32指令中的寄存器编码

Reg/基址/变址 <sup>①</sup> 字段	寄存器
0 0 0	EAX
0 0 1	ECX
0 1 0	EDX
0 1 1	EBX
1 0 0	ESP
1 0 1	EBP
1 1 0	ESI
1 1 1	EDI

① ESP (100)不能被用做变址寄存器。

771

寻址方式

表3-3列出了IA-32寻址方式，通常用汇编程序的语法来说明它们以及有效地址EA的产生方式。我们已经讨论了立即方式和使用ModR/M字节的Reg/OP码字段指明一个操作数位置的寄存器的情况。另外一个操作数的指定情况如表D-2所示。表中前4行列出了由ModR/M字节的2位Mod字段决定的间接寄存器、带位移的基址和寄存器寻址方式。3位R/M字段通常指明了这些方式中所涉及的寄存器（Reg）。除此之外的内容被用来产生表中所列出的其余寻址方式。除了直接方式以外所有方式都使用SIB字节，如图D-1c所示。SIB字节将基址和变址寄存器按表D-1进行编码。比例因子1，2，4，8的编码如表D-3所示。

正如表3-3指出的，ESP寄存器（编码100）不能被用做变址寄存器。事实上这并不是程序的限制，而是因为ESP将作为处理器堆栈指针。当位模式100存在于SIB字节的变址字段中时，不使用任何比例变址，但寻址方式的其他组成部分通常会正常地产生操作数的有效地址。这样做有如下好处。从表D-2可以看出，ESP不能在所列的前三种方式中使用，因为ESP编码（100）被用来表示产生表中后3种方式。但如果100同时存在于SIB字节的基址和变址字段中，由于不使用变址，可以使用ESP作为基址寄存器产生表中的前三个寻址方式。

注意为了产生带32位位移量的变址方式，编码为101的基址寄存器被用做带变址的基址寻址方式的异常（exception）。EBP寄存器还可以在带变址的基址方式中，以及在带有位移量的变址基址的方式中（只要位移量设成0）作为基址寄存器使用。

表D-2 由ModR/M和SIB字节选择的 IA-32寻址方式

ModR/M 字节		寻址方式
Mod 字段 <i>b<sub>7</sub> b<sub>6</sub></i>	R/M 字段 <i>b<sub>2</sub> b<sub>1</sub> b<sub>0</sub></i>	
0 0	Reg	间接寄存器 EA = [Reg]
0 1	Reg	带8位移量的基址寄存器 EA = [Reg] + Disp8
1 0	Reg	带32位移量的基址寄存器 EA = [Reg] + Disp32
1 1	Reg	寄存器 EA = Reg
异常		
0 0	1 0 1	直接寄存器 EA = Disp32
0 0	1 0 0	带变址的基址寄存器（使用SIB字节） EA = [Base] + [Index] × Scale  当Base=EBP寻址方式是： 变址带32位移量 EA = [Index] × Scale + Disp32
0 1	1 0 0	带变址和8位移量的基址寄存器（使用SIB字节） EA = [Base] + [Index] × Scale + Disp8
1 0	1 0 0	带变址和32位移量的基址寄存器（使用SIB字节） EA = [Base] + [Index] × Scale + Disp32

772

D.2 基本指令

经常使用的IA-32 指令在表D-4中按照字母顺序列出。除了在3.21.3节中的I/O块传送使用的跳转指令和有重写选项的特殊串指令外，表中包括了所有在第3章中使用过的指令。下面的两个小节描述了条件跳转指令和无条件跳转指令。串指令在D.4节中进行描述。表 D-4的第1列中给出了OP码助记符和指令名称。第2列指明了可以使用的操作数长度：B（字节）或者D（32位双字）。第3列给出了源操作数和目标操作数的可能位置，简写为：

- reg — 8个处理器寄存器之一
- mem — 一个存储单元
- imm — 一个8位或32位立即操作数
- imm8 — 一个8位立即操作数

表D-3 IA-32 SIB字节中比例字段编码

比例字段	比 例
0 0	1
0 1	2
1 0	4
1 1	8

773



表D-4 IA-32 指令

助记符 (名称)	长度	操作数		执行的操作	受影响的CC标志			
		dst	src		S	Z	O	C
ADC (带进位加)	B,D	reg reg mem reg mem	reg mem reg imm imm	$\text{dst} \leftarrow [\text{dst}] + [\text{src}] + [\text{CF}]$	x	x	x	x
ADD (加)	B,D	reg reg mem reg mem	reg mem reg imm imm	$\text{dst} \leftarrow [\text{dst}] + [\text{src}]$	x	x	x	x
AND (逻辑与)	B,D	reg reg mem reg mem	reg mem reg imm imm	$\text{dst} \leftarrow [\text{dst}] \wedge [\text{src}]$	x	x	0	0
BT (位检测)	D	reg reg mem mem	reg imm8 reg imm8	$\text{bit\#} = [\text{src}];$ $\text{CF} \leftarrow \text{bit\# of } [\text{dst}]$				x
BTC (位检测并求补)	D	reg reg mem mem	reg imm8 reg imm8	$\text{bit\#} = [\text{src}];$ $\text{CF} \leftarrow \text{bit\# of } [\text{dst}];$ complement bit# of [dst]				x
BTR (位检测和复位)	D	reg reg mem mem	reg imm8 reg imm8	$\text{bit\#} = [\text{src}];$ $\text{CF} \leftarrow \text{bit\# of } [\text{dst}];$ clear bit# of [dst] to 0				x

(续)

助记符 (名称)	长度	操作数		执行的操作	受影响的CC标志			
		dst	src		S	Z	O	C
BTS (位检测和置位)	D	reg reg mem mem	reg imm8 reg imm8	bit# = [src]; CF ← bit# of [dst]; set bit# of [dst] to 1				x
CALL (调用子程序)	D	reg mem		ESP ← [ESP] - 4; [ESP] ← [EIP]; EIP ← EA of dst				
CLC (清除进位)				CF ← 0				0
CLI (清除整数标志)				IF ← 0				
CMC (求补后进位)				CF ← $\overline{[CF]}$				x
CMP (比较)	B,D	reg reg mem reg mem imm imm	reg mem reg imm imm	[dst] - [src]	x	x	x	x
DEC (递减)	B,D	reg mem		dst ← [dst] - 1	x	x	x	
DIV (无符号除)	B,D		reg mem	对于B: [AL]/[src]; AL ← 商; AH ← 余数 对于D: [EAX]/[src]; EAX ← 商; EDX ← 余数	?	?	?	?

(续)

助记符 (名称)	长度	操作数		执行的操作	受影响的CC标志			
		dst	src		S	Z	O	C
HLT (暂停)				暂停操作直到产生复位或外部中断为止				
IDIV (有符号除)	B,D		reg mem	对于B: [AL]/[src]; AL ← 商; AH ← 余数 对于D: [EAX]/[src]; EAX ← 商; EDX ← 余数	?	?	?	?
IMUL (有符号乘)	B,D		reg mem	双倍长度乘积 对于B: AX ← [AL] × [src] 对于D: EDX,EAX ← [EAX] × [src]	?	?	x	x
	D	reg reg	reg mem	(单长度乘积) reg ← [reg] × [src]	?	?	x	x
IN (单独输入)	B,D	dst = AL 或 EAX src = imm8 或 [DX]		AL 或 EAX ← [src]				
INC (递增)	B,D	reg mem		dst ← [dst] + 1	x	x	x	
INT (软件中断)	D		imm8	Push EFLAGS; Push EIP; EIP ← 地址 (由imm8决定)				

(续)

助记符 (名称)	长度	操作数		执行的操作	受影响的CC标志			
		dst	src		S	Z	O	C
IRET (中断返回)	D			Pop EIP; Pop EFLAGS	x	x	x	x
LEA (加载有效地址)	D	reg	mem	reg $\leftarrow$ EA of src				
LOOP (循环)	D	target		ECX $\leftarrow$ [ECX] - 1; If ( [ECX] $\neq$ 0 ) EIP $\leftarrow$ target				
LOOPE (在相等/零时 循环)	D	target		ECX $\leftarrow$ [ECX] - 1; If ( [ECX] $\neq$ 0 ^ [Z] = 1 ) EIP $\leftarrow$ target				
LOOPNE (在不等/非零时 循环)	D	target		ECX $\leftarrow$ [ECX] - 1; If ( [ECX] $\neq$ 0 ^ [Z] $\neq$ 1 ) EIP $\leftarrow$ target				
MOV (移动)	B,D	reg reg mem reg mem	reg mem reg imm imm	dst $\leftarrow$ [src]				
MOVSX (符号扩展字节 到寄存器)	B	reg reg	reg mem	reg $\leftarrow$ 符号扩展				

(续)

助记符 (名称)	长度	操作数		执行的操作	受影响的CC标志			
		dst	src		S	Z	O	C
MOVZX (零扩展字节到寄存器)	B	reg reg	reg mem	reg ← 零扩展[src]				
MUL (无符号乘)	B,D		reg mem	(双倍长度乘积) 对于B: AX ← [AL] × [src] 对于D: EDX,EAX ← [EAX] × [src]	?	?	x	x
NEG (翻转)	B,D	reg mem		dst ← 补码 [dst]	x	x	x	x
NOP (无操作)				别名是: XCHG EAX,EAX				
NOT (逻辑补码)	B,D	reg mem		dst ← $\overline{[dst]}$				
OR (逻辑或)	B,D	reg reg mem reg mem imm mem imm	reg mem reg imm mem imm	dst ← [dst] ∨ [src]	x	x	0	0
OUT (单独输出)	B,D	dst = imm8 或 [DX] src = AL 或 EAX		dst ← [AL] 或 [EAX]				

(续)

助记符 (名称)	长度	操作数		执行的操作	受影响的CC标志			
		dst	src		S	Z	O	C
POP (弹出堆栈)	D	reg mem		$\text{dst} \leftarrow [[\text{ESP}]];$ $\text{ESP} \leftarrow [\text{ESP}] + 4$				
POPAD (除ESP外弹出 寄存器所有堆 栈)	D			从栈中弹出8个双字到 EDI, ESI, EBP, discard, EBX, EDX, ECX, EAX; $\text{ESP} \leftarrow [\text{ESP}] + 32$				
PUSH (压栈)	D		reg mem imm	$\text{ESP} \leftarrow [\text{ESP}] - 4;$ $[\text{ESP}] \leftarrow [\text{src}]$				
PUSHAD (将所有寄存器 压栈)	D			将 EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI 的内容压栈; $\text{ESP} \leftarrow [\text{ESP}] - 32$				
RCL (使用C标志循 环左移)	B,D	reg reg mem mem	imm8 CL imm8 CL	参见图2-32b; 源操作数 是循环计数的		?		x
RCR (使用C标志循 环右移)	B,D	reg reg mem mem	imm8 CL imm8 CL	参见图2-32d; 源操作数 是循环计数的		?		x
RET (从子程序返回)				$\text{EIP} \leftarrow [[\text{ESP}]];$ $\text{ESP} \leftarrow [\text{ESP}] + 4$				

(续)

助记符 (名称)	长度	操作数		执行的操作	受影响的CC标志			
		dst	src		S	Z	O	C
ROL (循环左移)	B,D	reg	imm8	参见图2-32a; 源操作数是循环计数的			?	x
		reg	CL					
		mem	imm8					
		mem	CL					
ROR (循环右移)	B,D	reg	imm8	参见图2-32c; 源操作数是循环计数的			?	x
		reg	CL					
		mem	imm8					
		mem	CL					
SAL (算术左移) 与SHL相同	B,D	reg	imm8	参见图2-30a; 源操作数是移位计数的	x	x	?	x
		reg	CL					
		mem	imm8					
		mem	CL					
SAR (算术右移)	B,D	reg	imm8	参见图2-30c; 源操作数是移位计数的	x	x	?	x
		reg	CL					
		mem	imm8					
		mem	CL					
SBB (借位减)	B,D	reg	reg	$\text{dst} \leftarrow [\text{dst}] - [\text{src}] - [\text{CF}]$	x	x	x	x
		reg	mem					
		mem	reg					
		mem	imm					
SHL (左移) 与SAL相同	B,D	reg	imm8	参见图2-30a; 源操作数是移位计数的	x	x	?	x
		reg	CL					
		mem	imm8					
		mem	CL					

(续)

助记符 (名称)	长度	操作数		执行的操作	受影响的CC标志			
		dst	src		S	Z	O	C
SHR (右移)	B,D	reg reg mem mem	imm8 CL imm8 CL	参见图2-30b; 源操作数是移位计数的	x	x	?	x
STC (置位进位标志)				$CF \leftarrow 1$				1
STI (置位中断标志)				$IF \leftarrow 1$				
SUB (减)	B,D	reg reg mem reg mem	reg mem reg imm imm	$dst \leftarrow [dst] - [src]$	x	x	x	x
TEST (检测)	B,D	reg mem reg mem	reg reg imm imm	$[dst] \wedge [src];$ 根据结果设置标志	x	x	0	0
XCHG (交换)	B,D	reg reg	reg mem	$[reg] \leftrightarrow [src]$				
XOR (异或)	B,D	reg reg mem reg mem	reg mem reg imm imm	$dst \leftarrow [dst] \oplus [src]$	x	x	0	0



在第4列中给出指令执行的操作。最后一列指明使用如下符号表示执行指令时条件码标志所受的影响：

- x — 受影响
- 0 — 置为0
- 1 — 置为1
- “空白” — 不受影响
- ？ — 不可预测

D.2.1 条件跳转指令

表D-5中列出了条件跳转指令。正如在第3.19.1节中讨论过的那样，目标地址直接在汇编语言程序中使用。实际上机器指令中包含着一个带符号的数（偏移量），它按字节方式指明了与当前指令指针寄存器相关的目标地址之间的距离。偏移量可以使用两种长度：1字节和4字节。当把一个汇编语言程序转变为机器语言时，汇编程序会计算偏移量。

表D-5 IA-32条件跳转指令

助记符	条件名称	条件码检测
JS	符号（负数）	SF = 1
JNS	无符号（正数或0）	SF = 0
JE/JZ	相等/为0	ZF = 1
JNE/JNZ	不等/不为0	ZF = 0
JO	溢出	OF = 1
JNO	不溢出	OF = 0
JC/JB	进位/无符号小于	CF = 1
JNC/JAE	无进位/无符号大于或等于	CF = 0
JA	无符号大于	CF ∨ ZF = 0
JBE	无符号小于或等于	CF ∨ ZF = 1
JGE	有符号大于或等于	SF ⊕ OF = 0
JL	有符号小于	SF ⊕ OF = 1
JG	有符号大于	ZF ∨ (SF ⊕ OF) = 0
JLE	有符号小于或等于	ZF ∨ (SF ⊕ OF) = 1

782

D.2.2 无条件跳转指令

在3.19.2节描述了无条件跳转指令JMP。和在条件跳转中使用的相对寻址方式一样，普通的寻址方式也可以用来指定目标地址。这为实现高级语言选择CASE语句中的多路分支提供了更大的便利。

D.3 前缀字节

如图D-1a所示，指令前缀字节被分成4组。指令中可以使用多个前缀字节。但是，每一组中一次只能使用一个字节。第一组包括重复字节码来表明指令操作需要重复一定的次数。具有这种选择的指令称为串指令，我们将在D.4节中描述。在3.21.3节中有一个在I/O设备和内存间进行双字块传送时重复串指令操作的例子。在3.23.3节、11.3.6节、11.3.7节中描述的SIMD流扩展（SSE）

指令也可以用这个组中的字节码表示。

有两个组，其中每组只包含一个字节码。正如D.5节中描述的，这些码用来替换缺省操作数大小或缺省地址大小。

第4组的前缀字节用来替换产生内存地址的段寄存器的缺省选择值。在11.3.1节中给出了段寄存器使用的概括性描述。

## D.4 其他指令

完整的IA-32指令集所包含的指令远比表D-4中列出的多得多。这里简要描述了表中没有包括的4种指令类型。

### D.4.1 串指令

对于在连续的存储单元中的数据项提供了专门的指令，可以高效地执行公共的重复操作。这些数据结构称作为串，相应的指令称作串指令。串中的每一项可以是字节或32位的双字。串指令可以用在将一个串从内存的一个区域传送到另一个区域的操作中，或者比较两个串是否相等。

我们利用串传送指令来说明串指令是如何执行的。操作码MOVSB和MOVSD用于传送字节和双字。这些指令与常规的传送指令不同，因为它们只包含OP码而没有明确的操作数。这时候该指令假设源操作数的地址在寄存器ESI中，目标操作数的地址在EDI中。MOVSB的执行包括将字节从源位置传送到目标位置，然后递增ESI和EDI指针寄存器。这个指令可以放在一个循环中来传送串的所有字节。还可以选择一个重复的前缀与指令一起来传送整个串。这样，除了初始化ESI和EDI寄存器外，还必须将ECX寄存器初始化成串的长度。每传送一个字节它的值就会被减1。指令

783

#### REP MOVSB

的执行传送了一个串的全部字节。这条指令被取出一次并且重复执行操作直到寄存器ECX 中的数值减到零为止。

提出带重复选择项的串指令是出于性能的考虑。使用下面的循环指令可以完成同样的工作

MOV BYTE PTR[EDI],[ESI]

指针寄存器不断增加，而计数寄存器ECX不断减少直到为0。但这种方法的执行时间会很长。

### D.4.2 浮点、MMX和SSE指令

IA-32指令中的IEEE格式（见第6章）提供了浮点数据操作的完整范围。图3-37中显示的8个浮点寄存器用来保存这些数据。除了加、减、乘、除操作外，还提供了三角函数运算操作。

3.23.2节中描述的MMX（多媒体扩展）指令，用来对存放于浮点寄存器或内存中的封装成64位的四倍长字的短整数，并行执行简单的算术和逻辑操作。在图形和信号处理应用中需要这样的操作。

784

SSE（流SIMD 扩展）指令首先在Pentium III处理器中使用并在Pentium 4中功能得到强化（见11.3.6节和11.3.7节），该指令对封装在8个128位处理器寄存器中的浮点数进行并行的算术操作。这些寄存器与通用寄存器及浮点寄存器不同。单独的数据项可以是32位或64位的浮点数。SEE指令对科学应用中的向量和矩阵运算十分有用。在Pentium 4增强版中，操作数还可以是64位的整数。

这些数据类型应用在数据安全应用中的加密和解密操作中。

## D.5 16位操作

在3.16.1节和11.3.2节中提到, IA-32处理器可以执行在早期的Intel处理器中使用16位地址和数据操作数方式的程序, 以及在本书中描述过的使用32位地址和数据操作数方式的程序。这两种方式均可以对字节操作数进行处理。当在32位操作方式下, 用操作码中的一位确定了操作数是一个字节还是一个32位的双字; 在16位方式下, 采用同样的位来判断操作数是一个字节还是一个16位字。操作的缺省方式由段描述符中的一位设定。11.3.2节对这些描述符做了简要的介绍。

在我们的讨论中, 默认处理器是在32位缺省方式下操作。但是, 当指令是前后相连的时候, 如果图D-1中所示的指令中的第一个字节使用了前缀字节, 那么在该指令的执行期间, 它的缺省方式将会被忽略。缺省操作数长度或缺省地址长度, 或者这两者一起都会由于不同的前缀字节的存在而被忽略。

## 785 D.6 编程实验

使用汇编语言程序进行实验的一个方便方式, 是使用高级语言提供的内连 (in-line) 汇编语言工具。第9章给出了一个在C程序中插入一段由汇编语言编写的I/O程序的例子。这里, 我们说明如何在C/C++中使用内连工具。微软公司提供了一个在由Intel IA-32处理器构造的个人电脑中的Windows操作系统环境下运行的编译器。

图D-2显示了图3-40a中的加法循环程序是如何合并到一个C/C++程序中的。汇编语言指令码缩写为结构

```
_asm{...}
```

C/C++程序的开头做了数据声明和初始化操作, 并且执行汇编语言程序的结果, 即内存单元SUM的值, 由程序末的输出语句打印出来。

程序中没有给出命名、打开源程序并进入、编译和执行的操作, 这是因为它们在很大程度上随所使用的软件环境不同而不同。

用汇编语言产生的一个16位机器指令列表, 如图D-2中的一个, 可以由编译器得出。理解图D-1中格式下IA-32指令的二进制编码的例子对学习列表很有帮助。4指令循环的列表如图D-3。用于对每条指令编码的字节的16进制表示在图D-3a的左边显示。图中b和c部分给出了ADD和JG指令的二进制编码细节。

首先, 考虑一下ADD指令。在操作码的右末位置1的两位含义如下: 最后一个1指明操作数大小是32位的; 倒数第二个1指明源操作数是储存在内存中的操作数。从表D-2中, 我们可以观察到ModR/M字节的Mod字段(00)和R/M字段(100)指明了内存操作数使用的是基址变址寻址方式, 而Reg/OP码字段(000)指定EAX寄存器是目标寄存器。SIB字节的基址字段(011)指明EBX为基址寄存器, 并且变址字段(111)指明EDI为变址寄存器。等级字段(10)选择4作为等级参数。

在图D-3c中编码的JG指令解释如下: 操作码的前4位(0111)指明了带有一个字节偏移量的条件跳转, 最后4位(1111)指定了“大于”条件。偏移量字节中保存的是-7的补码表示形式。它是紧跟在JG指令后的指令地址到循环开始处ADD指令地址之间的距离, 用字节表示。

```
# include <stdio.h>

void main(void)
{
    long NUM1[5];
    long SUM;
    long N;

    NUM1[0] = 17;
    NUM1[1] = 3;
    NUM1[2] = -51;
    NUM1[3] = 242;
    NUM1[4] = 113;
    SUM = 0;
    N = 5;

    .asm {
        LEA    EBX,NUM1
        MOV    ECX,N
        MOV    EAX,0
        MOV    EDI,0
    STARTADD:  ADD    EAX,[EBX + EDI*4]
                INC    EDI
                DEC    ECX
                JG     STARTADD
                MOV    SUM,EAX
    }

    printf ( "The sum of the list values is %ld \n" ,SUM);
}
```

786

图D-2 图 3-40a中的IA-32程序封装为C/C++程序

机器指令 (16进制)	汇编语言指令
03 04 BB	STARTADD: ADD EAX,[EBX + EDI*4]
47	INC EDI
49	DEC ECX
7F F9	JG STARTADD

a) 循环体编码

操作码	ModR/M字节	SIB字节
03 00000011	04 00 000 100	BB 10 111 011
ADD (双字)	(见表D-2)	(见图D-1c)

b) ADD指令

图D-3 图D-2中循环体的编码

操作码	偏移量
7F 01111111	F9 111111001
JG (短偏移量)	-7

c) JG指令

图D-3 (续)

787  
788

处理器堆栈的使用

编译器使用寄存器ESP和EBP分别作为处理器堆栈指针和帧指针。因此，内连式汇编语言程序不能将这些寄存器用做其他用途。而且，编译器分配内存变量，如在“主”程序中声明的NUM1、SUM和N作为栈的局部变量。当它们用在汇编语言的直接寻址方式中时，如图D-2中程序的前两条指令的情况，编译器采用基址加偏移量方式访问它们。帧指针EBP用作基址寄存器，并且压入堆栈的偏移量是负值，即从低位地址增长。如果这些变量在“主”程序之外声明，它们就成为全局变量并可用直接寻址方式来访问。

## 字符编码与数的转换

## E.1 字符编码

计算机中信息的存储和处理涉及使用若干个二进制变量对信息的各项进行编码的问题。在二进制系统中正数和负数表示的方法有些不同。第6章给出了最为常用的格式，同时还讨论了整数和浮点数。

对于主要处理商业数据的计算机来说，最常用的是表示和处理十进制的数。表E-1给出了单个数字最常用的编码，称为二进制编码的十进制数（BCD）。这种编码仅仅是4位二进制数字系统的前10个值（0~9）。这样的4位编码值的串只要使用一个适当的码作为符号位置，就可以用来代表所需范围内的任何正整数和负整数。

字母字符（A~Z）、操作符、标点符号、控制符号（+ - / ; LF CR EOT）和数字必须被表示成能进行文本存储和编辑，并适合高级语言输入、处理和输出操作的形式。能达到这种目的的两种标准码是ASCII码和EBCDIC码。标准ASCII码是一个7位码，EBCDIC码是一个8位码。表E-2和表E-3分别列出了标准ASCII码和EBCDIC码。其中ASCII码是目前最常用的格式。

在许多应用中比较偏好使用8位数；因此，基本的ASCII码通常被扩展为8位。常用的方法是将最高位，即第7位设置为0。另一个可行的方法是将第7位作为编码字符的校验位。

这里有必要对ASCII和EBCDIC码的结构做一些说明。注意在两种编码中十进制码（0~9）的低4位就是表E-1中的BCD码。这个特点简化了两种操作。第一，可以对表示为十进制数字的两个字符比较大小。这可以通过执行二进制数标准算法的同类逻辑电路来实现。当必须把十进制数按数字顺序存储时，这个操作是很有用的。第二，当一些输入串中作为一个单独的实体被存储和处理的表示十进制数的连续7或8位码由前后关系决定时，应用中有时去掉每个数字码最左边的3或4位，并将其表示成4位BCD码串。这个压缩（或称为数据打包）需要开始和结束分隔符，但是，当考虑到所需的存储空间时，这种做法是很恰当的。这也同样适用于字母字符编码。

表E-1 十进制数的BCD码

十进制数	BCD 码
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

## E.2 十进制数到二进制数的转换

这一节介绍了如何将一个定点的十进制数变成等价的二进制形式。二进制数

$$B = b_n b_{n-1} \cdots b_0 b_{-1} b_{-2} \cdots b_{-m}$$

V为B的十进制表示：

$$V(B) = b_n \times 2^n + b_{n-1} \times 2^{n-1} + \cdots + b_0 \times 2^0 \\ + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + \cdots + b_{-m} \times 2^{-m}$$

表E-2 7位ASCII码

位 位置	位位置 654							
3210	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	/	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	\$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	/	l	
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

NUL	空/空闲	SI	移进
SOH	头部开始	DLE	数据连接出口
STX	文本的开始	DC1-DC4	设备控制
ETX	文本的结尾	NAK	非应答
EOT	传输结束	SYN	同步空闲
ENQ	询问	ETB	传送的块结尾
ACK	应答	CAN	取消(数据错误)
BEL	监听信号	EM	媒介结尾
BS	退格	SUB	专用序列
HT	水平制表符	ESC	出口
LF	换行	FS	文件分隔符
VT	垂直制表符	GS	组分隔符
FF	换页	RS	记录分隔符
CR	回车	US	单元分隔符
SO	移出	DEL	删除/空闲

编码格式的位位置 = 

6	5	4	3	2	1	0
---	---	---	---	---	---	---

将一个定点的十进制数转换为二进制数时, 整数部分和小数部分要分别计算。首先, 整数部分的变换如下: 用2去除整数部分, 余数是二进制表示法中整数部分的最低位。商再被2除, 余数是二进制表示中的次低位。重复这个过程直至商为0 (包括商为0这一步)。

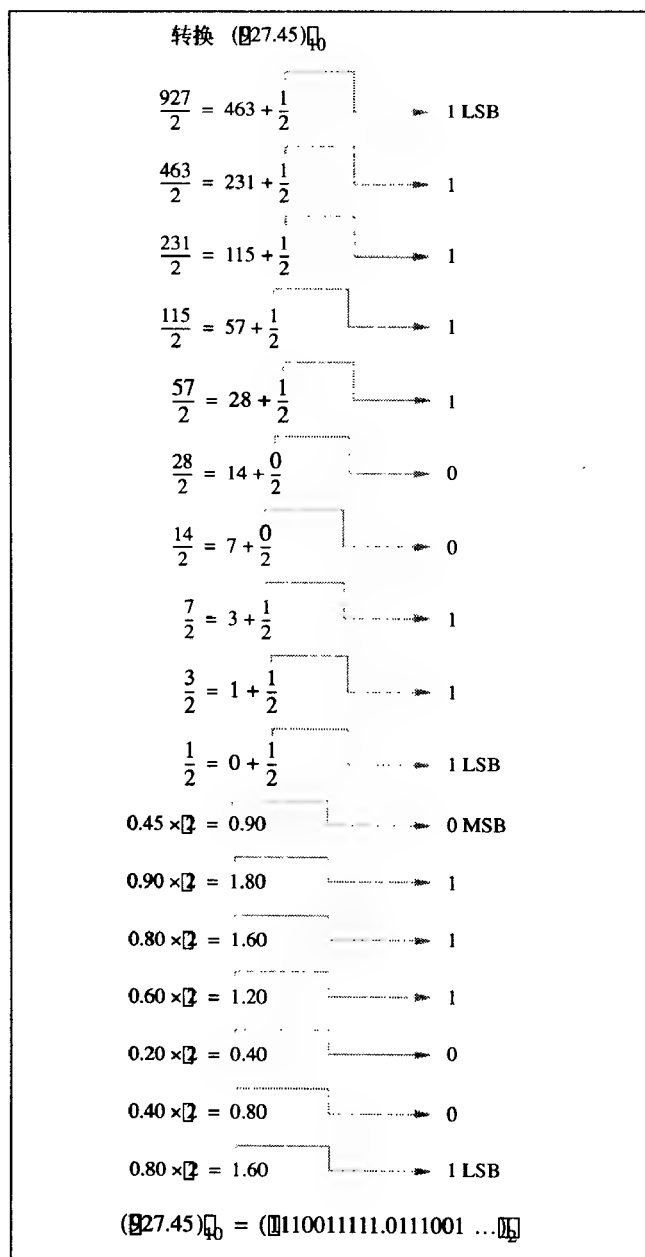
第二, 小数部分用乘2变换。乘积的结果在小数点左边的0或1用二进制中的一位来表示。乘积的小数部分再乘以2, 得到二进制表示的下一位。得到的第一位整数是小数点右边的第一位, 得到的下一位整数是小数点后的第二位, 依次类推。重复此过程直到达到要求的精确度为止。

图E-1给出了一个将  $(927.45)_{10}$  转换为二进制的例子。整数部分的转换总是精确的, 但一个精确的十进制小数对应的二进制小数未必是精确的。例如, 图E-1中的小数  $(0.45)_{10}$  并没有等价的二进制精确值。这从图中可以明显地看出来。在这种情况下, 二进制小数按一定的精确度标准取值。通常, 由  $k$  位小数表示产生的最大绝对误差  $e$  限制为  $e < 2^{-k}$ 。当然, 有些十进制小数有精确的二进制值。例如  $(0.25)_{10} = (0.01)_2$ 。





793  
794



图E-1 十进制数到二进制数的转换

# 索引

索引中的页码为英文原书页码, 与书中边栏页码一致。

## A

- Accumulator (累加器), 40
- Access time (访问时间)
  - magnetic disk (磁盘), 347
  - main memory (主存储器), 5
- Adder (加法器)
  - BCD (BCD), 405
  - carry-lookahead (超前进位), 374
  - circuit (电路), 370
  - full-adder (全加器), 368
  - half-adder (半加器), 404
  - propagation delay (传播延迟), 371
  - ripple-carry (行波进位), 368
- Addition (加法), 28, 368
  - carry (进位) 28, 368
  - carry-save (进位保留), 385
  - end-around carry (循环进位), 409
  - floating-point (浮点), 398
  - generate function (生成函数), 372
  - modular (模), 29
  - overflow (溢出), 32, 369
  - propagate function (传播函数), 372
  - sum (和), 28, 368
- Addition loop (加法循环)
- Address (地址) 5, 33
  - aligned/unaligned (可对齐/不可对齐), 36
  - big-endian (big-endian), 35
  - little-endian (little-endian), 35
- Address pointer (地址指针), 51
- Address space (地址空间), 33, 205
- Addressing mode (寻址方式), 47
  - absolute (绝对), 49
  - ARM (公司名, Advanced RISC Machine), 106
  - autodecrement (自动递减), 57, 69
  - autoincrement (自动递增), 57, 69
  - HP3000, 607
  - IA-32, 159, 772
  - immediate (立即), 49
  - index (变址), 52
  - indirect (间接), 50
  - PowerPC, 592
  - register (寄存器), 49
  - relative (相对), 56
  - 68000, 131
  - 68020, 583
- Advanced Micro Devices (AMD) (AMD公司), 591
- Alpha instructions (Alpha 指令), 596
- Alpha processors (Alpha 处理器), 597
  - 21064, 597
  - 21164, 597
  - 21264, 597
- Alphanumeric characters (字母数字字符), 33
- ASCII (ASCII), 791
- EBCDIC (EBCDIC), 792
- Altera Excalibur system (Altera Excalibur 系统), 547
- Amdahl's law (Amdahl 定律), 654
- Analog to digital (A/D) conversion (模/数转换), 515
- Annul bit (无效位), 491
- Apple computers (苹果计算机), 591, 594
- Arbitration (仲裁), 237
  - centralized (集中式), 237
  - distributed (分布式), 239
- Architecture (体系结构), 2
- Arithmetic and logic unit (ALU) (算术逻辑部件), 5
- ARM (公司名, Advanced RISC Machine)
  - addressing (寻址), 106
  - architecture versions (体系结构版本), 579
  - assembly language (汇编语言), 118
  - condition codes (条件码), 105, 117, 736
  - hard macrocell core (硬件宏单元核), 580
  - input/output (I/O) (输入/输出), 121
  - instructions (指令), 113, 734-750
    - conditional execution (条件执行), 106
    - operand shifting (操作数移位), 114, 737
    - pseudo-instructions (伪指令), 120
    - thumb (thumb), 579
  - interrupts (中断), 224
  - programming experiments (编程实验), 750
  - register (寄存器), 105
  - synthesizable core (综合核), 580
- ARM CPU cores (ARM CPU 内核), 581

ARM720T, 581  
 920T, 581  
 1020T, 581  
 StrongARM SA-110, 581  
 ARM processor cores (ARM处理器内核), 580  
   ARM7TDMI, 580  
   ARM9TDMI, 581  
   ARM10TDMI, 581  
 Array processor (阵列处理器), 620  
 ASCII code (ASCII码), 4, 791  
 Assembler (汇编程序), 58  
   two-pass (二遍扫描), 63  
 Assembler directives (commands) (汇编指示符(命令)), 60  
 Assembly language (汇编语言), 19, 58  
   ARM, 118  
   generic (通用), 58  
   IA-32, 170  
   mnemonics (助记符), 58  
   notation (标记), 19, 38  
   68000, 140  
   syntax (语法), 58  
 Associative search (相联检索), 318  
 Asynchronous bus (异步总线)  
 Asynchronous DRAM (异步DRAM), 299  
 Asynchronous transmission (异步传输), 566  
 Athlon processor (Athlon处理器), 591  
 Autovector (自动向量), 214

## B

Bandwidth (带宽)  
   communication (通信), 565, 567, 569, 570  
   memory (存储器), 304  
 Barrier synchronization (屏蔽同步), 650  
 Base register (基址寄存器), 55  
 Baseband (基带), 564  
 Baud rate (波特率), 564  
 Benchmark program (基准程序), 17, 656  
 Big-endian (big-endian), 35  
 Binary variable (二进制变量), 662  
 Binary-coded decimal (BCD) (二进制编码的十进制数), 4, 790  
   addition (加法), 405  
   packed (打包), 83  
 Bit (位), 4, 27  
 Bit map (位图), 559  
 Bit rate (位速率), 564  
 Bit-ORing (按位-或), 437  
 Booth algorithm (Booth算法), 380  
   bit-pair recoding (位偶重编码), 384

  skipping over 1s (跳过1位), 382  
 Booting (引导), 350  
 Branch (转移), 46  
   delay slot (转移延迟槽), 470, 491  
   delayed (延迟转移), 470  
   folding (转移重叠), 468  
   instruction (转移指令), 45  
   penalty (转移代价), 466  
   prediction (转移预测), 472, 491  
   target (转移目标), 46  
 Breakpoint (断点), 220  
 Bridge (桥), 259, 260, 262  
 Broadcast (广播), 633  
 Buffer (缓冲区)  
   circular (环形), 71, 531  
   register (寄存器), 10  
 Bus (总线), 9  
   arbitration (仲裁), 237  
   asynchronous (异步), 244  
   cycle (周期), 241  
   master (主控), 237  
   propagation delay (传播延迟), 241  
   scheduling (调度) 10, 624  
   skew (相位偏移), 245  
   synchronous (同步), 241, 264  
   timing (时序), 242  
 Bus standards (总线标准), 259  
   ISA, 260  
   PCI, 261  
   SCSI, 266  
   USB, 272  
   X3.131 (SCSI), 266  
 Byte (字节), 33  
 Byte addressable (按字节寻址), 35

## C

Cache memory (高速缓存存储器), 5, 13, 294, 313-329  
   in ARM710T, 326  
   associative mapping (相联映射), 318  
   block (块), 315  
   coherence (一致性), 321, 641  
   direct mapping (直接映射), 317  
   dirty bit (脏位), 316, 326  
   hit (命中), 316  
   hit rate (命中率), 332  
   levels (层次), 313, 335  
   line (块), 315  
   load through (直接装入), 316  
   lockup-free (无锁定), 337

mapping function (映射功能), 316  
miss (失效), 316  
miss penalty (失效开销), 332  
miss rate (失效率), 332  
in Pentium III, 326  
in Pentium4, 329  
in 68040, 325  
replacement algorithm (替换算法), 316, 321  
set-associative mapping (组相联映射), 318  
snoopy controls (监听控制), 643  
tag (标志), 318  
valid bit (有效位), 319, 326  
write back (写回), 316, 642  
write buffer (写缓冲区), 335  
write through (直接写), 316, 642  
Carry (进位), 28, 368  
flag (标志)  
Cartridge tape (盒式磁带), 359  
Cathode-ray tube (CRT) (阴极发光管), 558  
CC-NUMA systems (CC-NUMA系统), 645  
CD-ROM (只读光盘), 5, 355  
Character codes (字符码), 33  
ASCII, 791  
EBCDIC, 792  
Character string (字符串), 36  
Charge-coupled device (CCD) (电荷耦合器), 557  
Circuit switching (线路交换), 635  
Circular buffer (queue) (环形缓冲区(队列)), 533  
CISC (Complex Instruction Set Computer) (CISC (复杂指令集计算机)), 17, 97  
Clock (时钟), 14  
cycle (周期), 14  
rate (速率), 16  
Clock recovery (时钟复位), 565  
Coherence (一致性)  
Combinational circuits (组合电路), 691  
Compiler (编译程序, 编译器), 4, 11  
optimizing (优化编译器), 17  
Complement (补码), 665  
Complementary metal-oxide semiconductor (CMOS) (互补金属氧化物半导体), 681  
Complex Instruction Set Computer (复杂指令集计算机)  
Complex programmable logic device (CPLD) (复杂可编程逻辑器件), 711  
Computer (计算机), 2  
Computer-aided design (CAD) (计算机辅助设计), 677  
Concurrency (并发性), 15  
Condition code register (条件码寄存器), 46  
Condition codes (条件码), 46, 84  
ARM, 105, 117, 736

IA-32, 156, 171, 783  
side effect (副作用), 478  
68000, 141, 768  
Conditional branch (条件转移), 46  
Context switch (上下文切换), 223  
Control store (控制存储器), 430  
Control unit (控制单元), 6  
Control word (控制字), 430  
Coprocessor (协处理器)  
ARM instructions (ARM指令), 750  
Counter (计数器)  
ripple (行波), 702  
synchronous (同步), 703, 714  
Critical section (临界区), 640  
Crossbar (纵横), 625  
Cycle stealing (周期挪用), 237  
Cyclic redundancy check (CRC) (循环冗余校验), 279

## D

Daisy chain (菊花链), 216  
Data (数据), 4  
Data communication equipment (数据通信设备), 564  
Data striping (数据条带化), 351  
Data terminal equipment (数据终端设备), 564  
Data types (数据类型)  
bit (位), 27  
byte (字节), 33  
character (字符), 33  
floating-point (浮点), 394  
fraction (小数), 394  
integer (整型, 整数), 27  
string (串), 36  
word (字), 33  
Datapath (数字通路), 414  
De Morgan's rule (德摩根律), 670  
Debouncing (反弹), 249  
Debugging (调试), 63, 219  
Decoder (译码器), 703  
Demodulation (解调), 564  
Desktop computer (台式机), 2  
Device driver (设备驱动程序), 223  
Differential signaling (微分信号), 308  
Digit packing (数字打包)  
Digital camera (数码照相机), 514  
Digital subscriber loop (DSL) (数字式用户环路), 569  
Direct memory access (直接存储器访问)  
Directory-based cache coherence (基于目录的高速缓存一致性), 643  
Dirty bit (脏位)  
Disk (盘)

Disk array (磁盘阵列), 351  
 Dispatch operation (调度操作), 467, 486  
 Dispatch unit (调度部件)  
 Display buffer (显示缓冲区), 559  
 Displays (显示器), 558  
 Distributed computer system (分布式计算机系统), 618  
 Distributed computing (分布式计算), 21  
 Distributed memory system (分布式存储器系统), 623  
 Division (除法), 390  
   floating-point (浮点), 398  
   nonrestoring (不恢复), 391  
   restoring (恢复), 391  
 DMA (Direct Memory Access) (直接存储器访问), 235  
   block (burst) mode (块(脉冲)模式), 237  
   controller (控制器), 235  
 Don't-care condition (无关项条件), 674  
 Dot product (点积)  
 DVD (数字多功能光盘), 357  
 Dynamic memory (DRAM) (动态存储器 (DRAM)), 299

## E

Echoback (回显), 207  
 Edge-triggered flip-flop (边沿触发触发器), 694  
 Effective address (有效地址), 50  
 Effective throughput (有效吞吐量), 624  
 Elapsed time (消耗时间), 13  
 Embedded processor (嵌入式处理器), 517  
 Embedded system (嵌入式系统), 512  
 Emulation (仿真), 445  
 Enterprise system (企业系统), 2  
 Ethernet (以太网), 646  
 Exception (异常)  
   floating-point (浮点), 397  
   imprecise (不精确), 484  
   precise (精确的), 485  
 Execution phase (执行阶段), 413  
 Eye pattern (眼孔图), 568

## F

Fan-in (扇入), 687  
 Fan-out (扇出), 687  
 Fat-tree network (胖树网络), 631  
 Fetch phase (取指令阶段), 413  
 Field-programmable gate array (FPGA) (现场可编程门阵列) 547, 712  
 FIFO (first-in, first-out) queue (FIFO (先入先出) 队列) 71, 531  
 File (文件), 11

File mark (文件标记), 358  
 File server (文件服务器), 2  
 Finite state machine (有限状态机), 719  
 Flash memory (闪存), 312  
 Flat panel display (平面显示器), 560  
 Flip-flops (触发器), 690-699  
   D (D触发器), 694-697  
   edge-triggered (边沿触发), 694  
   gated latch (门控锁存器), 690  
   JK (JK触发器), 697  
   latch (锁存器), 690  
   master-slave (主从), 694  
   SR latch (SR锁存器), 690  
   T (T触发器), 697  
 Flit (flit), 634  
 Floating point (浮点), 393  
   addition-subtraction unit (加法/减法部件), 400  
   arithmetic operation (算术运算), 398  
   chopping (截断), 399  
   coprocessor (协处理器), 588, 750  
   double precision (双精度), 396  
   exception (异常), 397  
   exponent (指数), 394  
     excess-x representation (余x表示), 396  
   format (格式), 395  
   guard bits (保护位), 399  
   IEEE standard (IEEE标准), 394  
   mantissa (尾数), 394  
   normalization (规格化), 394  
   overflow (溢出), 396  
   representation (表示), 395  
   rounding (舍入), 399  
   scale factor (比例因子), 394  
   significant digits (有效数字), 394  
   single precision (单精度), 396  
   special values (特殊值), 397  
     denormal (非规格化), 397  
     Not a Number (NaN) (非数, 无定义数), 397  
   sticky bit (粘着位), 400  
   truncation (截取), 399  
     biased/unbiased (有偏/无偏), 399  
   underflow (下溢), 396  
 Floppy disk (软盘), 350  
 Four-wire link (四线连接), 565  
 Frame pointer (结构指针), 77  
 Frequency-shift keying (FSK) (移频键控), 564  
 Full-duplex (FDX) link (全双工 (FDX) 连接), 565  
 Fully-connected network (全连接网), 625

## G

Gated latch (门控锁存器), 690  
 General-purpose register (通用寄存器), 8  
 Global address space (全局地址空间), 637  
 GPU (Graphics Processing Unit) (图形处理单元), 561  
 Graphics (图形)  
   accelerator (加速卡), 561  
   port (端口), 562  
   processing unit (处理单元), 561  
 Gray code (Gray码), 726

## H

Half-duplex (HDX) link (半双工连接), 565  
 Handshake (握手), 244  
 Hardware (硬件), 2  
 Hardwired control (硬件控制), 425  
 Hertz (赫兹), 14  
 Hexadecimal (十六进制)  
 High-level language (高级语言), 4, 11, 26  
 Hold time (保持时间), 697  
 HP3000  
   addressing (寻址), 607  
   instructions (指令), 606  
   stack structure (栈结构), 604  
 Hybrid (混合器), 566  
 Hypercube network (超立方体网络), 628

## I

Index register (变址寄存器), 52  
 Input unit (输入设备), 4  
 Input/output (I/O) (输入/输出)  
   address space (地址空间), 205  
   instructions in IA-32 (IA-32的指令), 175  
   interface circuit (接口电路), 206, 248, 259  
   memory-mapped (存储器映射), 66, 204  
   program-controlled (程序控制), 64, 207  
   register (寄存器), 206  
   status flag (状态标志), 67, 206  
   unit (部件), 3  
 Input/output device (输入/输出设备)  
 Input/output port (输入/输出端口), 248  
   bidirectional (双向), 254  
   parallel (并行), 248, 518  
   serial (串行), 257, 521  
 Instruction (指令), 3, 37  
   commitment (提交), 485  
   encoding (编码), 94  
   execution phase (执行阶段), 413

  dispatch (调度), 467, 486  
   fetch phase (取指令阶段), 413  
   fields (域, 字段), 62, 95  
   grouping (分组), 496  
   hazards (阻塞), 504  
   operands (操作数), 39  
   privileged (特权), 220, 343  
   queue (队列), 467  
   reordering (重排序), 471  
   retired (释放), 485  
   side effects (副作用), 478  
   synthetic (综合), 489  
 Instruction encoding (指令编码), 94  
 Instruction format (指令格式)  
   ARM, 106, 735  
   HP3000, 606  
   IA-32, 168, 770, 771  
   IA-64, 598  
   one-address (单地址), 40  
   68000, 137  
   three-address (三地址), 39  
   two-address (两地址), 39  
   zero-address (零地址), 42  
 Instruction register (IR) (指令寄存器), 7, 43, 412  
 Instruction set architecture (ISA) (指令集体系结构), 26  
 Instruction unit (指令部件), 3  
 Instructions (指令)  
   arithmetic (算术), 7, 38, 70  
   branch (分支), 45  
   data transfer (数据传送), 38  
   input/output (输入/输出), 66  
   logic (逻辑), 81  
   shift and rotate (移位和循环移位), 82  
   subroutine (子程序), 72  
 Integrated circuit (IC) (集成电路), 16, 20, 688  
 Intel IA-32 Pentium (Intel IA-32 Pentium)  
   addressing modes (寻址方式), 159, 772  
   assembly language (汇编语言), 170  
   condition codes (条件码), 156, 171, 783  
   input/output (I/O) (输入/输出), 174  
     block transfers (块传送), 176  
     isolated (独立), 175  
     memory-mapped (存储器映射), 174  
   instructions (指令), 164, 171, 182, 773-784  
     encoding (编码), 770, 771  
     format (格式), 168, 770, 771  
     multimedia (MMX) (多媒体扩展), 183  
     string (串), 176, 783  
     vector SIMD (SSE) (向量SIMD扩展), 184

- interrupts (中断), 231-234
  - memory segmentation (存储器分段), 586
    - protected mode (保护模式), 587
    - real mode (实模式), 586
    - segment registers (段寄存器), 156, 586
  - programming experiments (编程实验), 785
  - register structure (寄存器结构), 156
  - sixteen-bit mode (16位模式), 588, 785
  - subroutines (子程序), 177
    - stack frame (栈结构), 179
  - Intel IA-64
    - instructions (指令), 598
      - bundles (包), 598
      - conditional execution (条件执行), 598
    - Itanium processor (Itanium处理器), 602
    - register rotation (寄存器循环), 602
    - register stack (寄存器堆栈), 600
    - register windows (寄存器窗口), 602
    - speculative loads (推测性装入), 600
  - Intel processors (Intel 处理器)
    - 8051, 542
    - 8086, 585
    - 8088, 585
    - 80286, 585
    - 80386, 588
    - 80486, 588
    - Itanium, 602
    - Pentium, 589
    - Pentium II, 590
    - Pentium III, 590
    - Pentium 4, 590
    - Pentium pro, 589
  - Intellectual property (IP) (知识产权), 547
  - Intellimouse (智能鼠标), 555
  - Interconnection network (互连网络), 622-636
    - crossbar (纵横交叉), 625
    - fat tree (胖树), 631
    - hypercube (超立方体), 628
    - mesh (网状), 630
    - multistage (多段), 626
    - ring (环状), 631
    - shuffle network (混洗网), 627
    - single-bus (单总线), 624
    - tree (树状), 630
    - torus (圆环形), 630
  - Interleaving (交叉), 330
  - Internet (Internet), 2
  - Interrupt (中断), 9
    - acknowledge (确认), 214
    - in ARM, 224-229
    - disabling (禁止), 212
    - edge-triggered (边沿触发), 229, 212
    - enabling (允许), 212
    - in IA-32, 231-234
    - latency (等待), 210
    - mask (屏蔽), 224
    - nesting (嵌套), 213
    - nonmaskable (不可屏蔽), 229
    - priority (优先级), 215
    - service routine (服务程序), 9
    - in 68000, 229-231
    - software (软件), 220
    - vectored (向量), 214
  - IR (instruction register) (指令寄存器), 7, 43, 412
  - ISA (instruction set architecture) (指令集体系结构), 26
  - Isochronous (等时), 273, 281
  - Itanium processor (Itanium处理器), 602
- J**
- Joystick (操作杆), 556
  - JTAG port (JTAG端口), 712
- K**
- Karnaugh map (卡诺图), 671
  - Keyboard (键盘), 554
- L**
- Laser printer (激光打印机), 560
  - Latch (锁存器), 690
  - LIFO (last-in, first-out) (后进先出), 68
  - Link register (链接寄存器), 72
  - Linked list (链表), 90
    - insertion/deletion (插入/删除)
  - Liquid crystal display (液晶显示器), 559
  - Little-endian (little-endian), 35
  - Load through (直接装入)
  - Loader (装载程序), 63
  - Load-store multiple operands (Load-store多操作数)
    - ARM, 112
    - IA-32, 783
    - 68000, 146
  - Local area networks (LAN) (局域网), 646
    - Ethernet (以太网), 646
    - token ring (令牌环), 647
  - Locality of reference (引用局部性), 315
  - Lock (锁), 640
  - Logic circuits (逻辑电路), 662-724
  - Logic families (逻辑系列)

CMOS, 681  
Logic function (逻辑函数), 662-677  
    AND (与), 664  
    EXCLUSIVE-OR (XOR) (异或), 664  
    minimization (最小化), 668  
    NAND (与非), 674  
    NOR (或非), 674  
    NOT (非), 665  
    OR (或), 662  
    synthesis (组合), 664  
Logic gates (逻辑门), 662  
    fan-in (扇入), 687  
    fan-out (扇出), 687  
    noise margin (噪声容限), 686  
    propagation delay (传播延迟), 686  
    threshold (阈值), 678, 684  
    transfer characteristic (传输特性), 684  
    transition time (转换时间), 686  
Logical address (逻辑地址), 294, 338  
Long-haul networks (远程网), 646  
Loop (循环)  
    with branch prediction (带转移预测), 473, 492  
    with delayed branch (带延迟转移), 470, 492  
Loosely coupled multicomputer (松散耦合多计算机), 645  
LRU (least-recently used) replacement (最近最少使用替换), 321

## M

Machine instruction (机器指令), 3, 94  
Machine language (机器语言), 4, 19, 26  
Magnetic disk (磁盘), 5, 344-352  
    access time (访问时间, 存取时间), 347  
    controller (控制器), 348  
    data buffer/cache (数据缓冲区/高速缓存), 348  
    data encoding (数据编码), 344  
    drive (驱动器), 346  
    floppy disk (软盘), 350  
    latency (等待), 347  
    organization (组织结构), 346  
    rotational delay (旋转延迟), 347  
    seek time (寻道时间), 347  
    Winchester (温切斯特), 345  
Magnetic tape (磁带), 358  
    cartridge (盒式磁带), 359  
    format (格式化), 358  
Mailbox memory (邮箱式存储器), 659  
Mainframe (主机), 2  
Manchester code (曼彻斯特码), 344  
Master-slave (主从)  
Mechanical computing devices (机械计算设备), 19  
Memory (存储器), 4  
    access time (访问时间), 5, 294  
    address (地址), 5, 33  
    address register (MAR) (地址寄存器), 8, 293  
    address space (地址空间), 292  
    asynchronous DRAM (异步DRAM), 299  
    bandwidth (带宽), 304  
    bit line (位线), 295  
    byte addressable (按字节寻址), 35  
    cache (高速缓存)  
    cell (单元), 298, 310  
    controller (控制器), 307  
    cycle time (周期时间), 294  
    data register (MDR) (数据寄存器), 8, 293  
    DDR SDRAM, 304  
    DIMM, 306  
    dynamic (DRAM) (动态RAM), 299  
    fast page mode (快速页模式), 301  
    hierarchy (层次结构), 5, 313  
    interleaving (交叉), 330  
    latency (延迟), 304  
    main (主要的), 5, 13, 313  
    multiple module (多模块), 330  
    Rambus (Rambus), 308  
    random-access memory (RAM) (随机访问存储器), 5, 294  
    read cycle (读周期), 9  
    read-only memory (ROM) (只读存储器), 310  
    refreshing (刷新), 301, 308  
    RIMM, 309  
    SIMM, 306  
    static (SRAM) (静态随机存储器), 297, 305  
    synchronous DRAM (SDRAM) (同步动态随机存储器), 302  
    unit (部件), 4  
    word (字), 5, 33  
    word length (字长), 33, 292  
    word line (字线), 295  
    write cycle (写周期), 9  
Memory management unit (MMU) (存储器管理部件), 294, 339  
    in ARM, 581  
    in 680X0, 584  
    in 80X86, 586  
Memory pages (存储器页), 133  
Memory segmentation (存储器分段), 586  
Memory-mapped I/O (存储器映射I/O), 66, 204  
Mesh network (网状网络), 630  
Message-passing (消息传递), 19, 645, 651



protocol (协议), 623

MFLOPS measure (MFLOPS度量), 656

Microcontroller (微控制器), 512, 518-521, 541

- ARM, 543
- Intel, 542
- Motorola, 542

Microinstruction (微指令), 430

- fields (域/字段), 433
- horizontal (横向), 434
- sequencing (顺序), 437, 440
- vertical (纵向), 434

Microoperation (微操作), 433

Microprogram counter (微程序计数器), 430

Microprogram memory (微程序存储器)

Microprogrammed control (微程序控制), 429

Microroutine (微程度), 430

Microwave oven (微波炉), 512

MIMD system (多指令流多数据流系统), 620

Miss (失效)

Mnemonic (助记符), 58

Modem (调制解调器), 564

- cable (电缆), 569

Modulation (调制), 564

Motherboard (主板), 259, 305

Motorola processors (Motorola处理器)

- ColdFire family (ColdFire系列), 585
- ColdFire MCF5xxx (ColdFire MCF5xxx), 543
- 68000, 130
- 68020, 582
- 68030, 584
- 68040, 584
- 68060, 585
- 68HC11, 542
- 683xx, 543

Motorola 68000, 130

- addressing modes (寻址方式), 131, 752
- assembly language (汇编语言), 140
- condition codes (条件码), 141, 768
- input/output (I/O) (输入/输出), 145
- instruction set (指令集), 136, 141, 151, 754-767
- interrupts (中断), 229-231
- register structure (寄存器结构), 131
- subroutines (子程序), 146

Mouse (鼠标), 555

Multicast (多播), 633

Multicomputer (多计算机), 18, 638, 645

Multiple issue (多发操作), 482

Multiplexer (多路复用器), 705

Multiplication (乘法), 376

- array implementation (阵列实现), 376

- Booth algorithm (Booth算法), 380
- carry-save addition (进位保留加法), 385
- fast (快速), 383
- floating-point (浮点), 398
- sequential implementation (顺序实现), 378
- signed-operand (有符号操作数), 380

Multiprocessor (多处理器), 18, 618, 622

- cache coherence (高速缓存一致性), 641
- caches (高速缓存), 637
- global memory (全局存储器), 623
- interconnection network (互连网络), 624-636
- local memory (局部存储器), 623
- private address space (私有地址空间), 638
- program parallelism (程序并行性), 638
- shared memory (共享存储器), 18, 637
- shared variables (共享变量), 640
- speedup (加速), 653

Multiprogramming (多道程序), 12

Multistage network (多段网络), 626

Multitasking (多任务), 12, 221

## N

Network of workstations (工作站网络), 647

Noise (噪声), 686

Noise margin (噪声容限), 686

Nonblocking switch (无阻塞交换), 626

Notebook computer (笔记本电脑), 2

NUMA multiprocessors (NUMA多处理器), 623

Number conversion (数字转换), 793

Number representation (数的表示)

- floating-point (浮点), 394
- hexadecimal (十六进制), 64
- 1's-complement (反码), 27
- sign and magnitude (原码), 27
- signed integer (有符号整数), 28
- ternary (三进制), 404
- 2's-complement (补码), 27

## O

Object program (目标程序), 4, 58

1's-complement representation (反码表示), 27

OP code (操作码), 59, 95

- in ARM, 106
- in IA-32, 168
- in 68000, 135

Open-drain (漏极开路), 211

OpenGL (开放式图形库), 563

Operand forwarding (操作数传递), 462

Operand (操作数), 26

Operating system (操作系统), 11

- multitasking (多任务), 221
- process (进程), 221
- scheduling (调度), 221
- Optical disk (光盘), 5, 352-358
  - CD-Recordable (可刻录CD), 356
  - CD-Rewritable (可擦写CD), 356
  - CD-ROM (只读光盘), 355
  - DVD (数字多功能光盘), 357
- Output unit (输出设备), 6
- Overflow (溢出), 32, 84
  - flag (标志)

## P

- Packet (包), 624
- Page (页), 339
- Parallel I/O port (并行I/O端口), 248, 518
- Parallel processing (并行处理), 619
- Parameter passing (参数传递), 74
  - by reference (按地址), 75
  - by value (按值), 75
- Parity (奇偶校验), 567
- PC (程序计数器)
- Performance (性能), 13
  - basic equation (基本公式), 14
  - execution time (执行时间), 13
  - measurement (测量), 17, 656
  - memory (存储器), 329
  - multiprocessors (多处理器), 653
  - pipeline (流水线), 503-505
  - processor (处理器), 13
- Peripheral device (外围设备), 554
- Personal computer (个人计算机), 2
- Phase encoding (相位编码), 344
- Phase-shift keying (PSK) (移相键控), 564
- Physical address (物理地址), 338
- Pipelining (流水线), 15, 454
  - addressing modes (寻址方式), 465, 476
  - basic concepts (基本概念), 454
  - branching (转移), 466
  - bubbles (气泡), 460
  - condition codes (条件码), 465, 478
  - data hazards (数据阻塞), 459, 461
  - instruction hazards (指令阻塞), 459, 465
  - performance (性能), 503
  - stalling (拖延), 459
  - structural hazard (结构阻塞), 460
- Pixel (像素), 558
- Plasma display (等离子显示器), 560
- Plug-and-play (即插即用), 261, 265, 274
- Pointer register (指针寄存器), 51
- Polling (轮询), 207, 213
- Pop operation (出栈操作)
- Port (端口)
- PowerPC
  - addressing modes (寻址方式), 592
  - instructions (指令), 592
  - registers (寄存器), 591
- PowerPC processors (PowerPC 处理器)
- Prefetching (预取), 336
- Primary storage (主存储器), 5
- Printer (打印机), 6, 560
- Priority (优先级) 215
- Private memory (私有存储器), 623
- Privileged instruction (特权指令), 220, 343
- Process (进程)
- Processor (处理器), 3
- Processor stack (处理器堆栈), 73
- Processor status word (PSW) (处理器状态字), 46
  - in ARM, 224
  - in IA-32, 232
  - in 68000, 230
- Processor time (处理器时间), 13
- Program (程序), 4
- Program counter (PC) (程序计数器), 8, 43, 412
- Program examples (程序示例)
  - addition loop (加法循环)
    - ARM, 118
    - generic (通用), 45, 51, 57
    - IA-32, 166
    - 68000, 143
  - character transfer (字符传送), 525
  - circular buffer (环形缓冲区), 531
  - digit packing (数字打包)
    - ARM, 116
    - generic (通用), 82
    - IA-32, 174
    - 68000, 151
  - dot product (点积)
    - ARM, 126
    - generic, 86
    - IA-32, 184
    - 68000, 152
  - message-passing multiprocessor (消息传递多处理器), 651
  - shared-memory multiprocessor (共享存储器多处理器), 648
- input/output (输入/输出)
  - ARM, 121
  - generic, 67, 207
  - IA-32, 175
  - 68000, 145

interrupt service routine (中断服务程序)  
 ARM, 230  
 generic, 219  
 IA-32, 234  
 68000, 232

linked-list insertion/deletion (链表插入/删除)  
 ARM, 127  
 generic, 92, 93  
 IA-32, 185  
 68000, 154

reaction timer (反应计时器), 537

sorting (排序)  
 ARM, 127  
 generic, 87  
 IA-32, 185  
 68000, 153

subroutines (子程序)  
 ARM, 122  
 generic, 74, 76, 79  
 IA-32, 177  
 68000, 146

Program parallelism (程序并行性), 638  
 shared variables (共享变量), 640

Program state (程序状态), 221

Program-controlled I/O (程序控制I/O)

Programmable array logic (PAL) (可编程阵列逻辑), 710

Programmable logic array (PLA) (可编程逻辑阵列), 707

Programming experiments (编程实验)  
 in ARM, 750  
 in IA-32, 785

Propagation delay (传播延迟)

Protected mode (保护模式), 587

Protection (保护), 343

Push operation (进栈操作)

## Q

Quadrature amplitude modulation (QAM) (正交调幅), 564

Queue (队列), 71

## R

RAID disk systems (RAID磁盘系统), 351

Rambus memory (Rambus存储器), 308

Random-access memory (RAM) (随机访问存储器), 5

Raster scan (光栅扫描), 558

Reaction timer (反应计时器), 635

Reactive system (反应系统), 541

Read-modify-write (读-修改-写), 640

Read-only memory (ROM) (只读存储器), 310  
 electrically erasable (EEPROM) (电可擦除可编程只读存储器), 311  
 erasable (EPROM) (可擦除可编程只读存储器), 311  
 flash (闪存), 312  
 programmable (PROM) (可编程只读存储器), 311

Real mode (实模式), 586

Real-time processing (实时处理), 210, 535

Reduced Instruction Set Computer (精简指令集计算机)

Refreshing (刷新)

Register (寄存器), 6, 699  
 base (基址), 55  
 general-purpose (通用), 8  
 index (变址), 52  
 port (端口), 423  
 renaming (重命名), 485, 498  
 windows (窗口), 488

Register transfer notation (RTN) (寄存器传送标记), 37

Rendering (渲染), 562

Reorder buffer (重新排序缓冲器), 485

Replacement algorithm (置换算法), 321

Ring network (环状网络), 631

RISC (Reduced Instruction Set Computer) (精简指令集计算机) 17, 97, 578

Rounding (舍入)

## S

Scalability (可伸缩性), 633

Scaler (定标器), 702

Scanner (扫描仪), 557

Scheduling (调度)

SCI Standard (SCI标准), 644

SCSI bus (SCSI总线)

Secondary cache (二级高速缓存), 314

Secondary storage (辅助存储器), 5, 344-359

Segmentation (分段)

Sequential circuits (时序电路), 691, 714-723  
 finite state machine (有限状态机), 719  
 state assignment (状态分配), 716  
 state diagram (状态图), 715  
 state table (状态表), 716  
 synchronous (同步), 718

Serial I/O interface (串行I/O接口), 257, 521

Serial transmission (串行传输), 563

Server (服务器), 2, 512

Setup time (建立时间), 242, 697

Seven-segment display (七段显示器), 704

Shared memory (共享存储器), 18, 637, 648

- Shared variables (共享变量), 640
    - busy-waiting (忙等待), 658
    - critical section (临界区), 640
    - locks (锁), 640
    - read-modify write access (读-修改-写访问), 640
    - Test-and-Set instruction (测试-置位指令), 640
  - Shift register (移位寄存器), 700
  - Shuffle network (混洗网), 627
  - Side effects (副作用)
  - Sign bit (符号位), 27, 34
  - Sign extension (符号扩展), 32
  - SIMD system (单指令流多数据流系统), 620
  - Simplex link (单工连接), 565
  - Simulated annealing (模拟退火), 655
  - SISD system (单指令流单数据流系统), 619
  - Skew (偏斜)
  - Small-scale integration (SSI) (小规模集成SSI), 689
  - Snoopy cache (监听高速缓存), 643
  - Soft processor core (软处理器核), 547
  - Software interrupts (软件中断)
  - Sorting (排序)
  - Source program (源程序), 4, 58
  - SP (堆栈指针)
  - SPMD application (SPMD应用), 651
  - SPEC rating (SPEC等级), 18
  - Speedup (加速), 653
  - Split-transaction protocol (分割事务协议), 624
  - Stack (栈, 堆栈), 68
    - in ARM, 122
    - frame (结构)
      - ARM, 123
      - generic, 75
      - IA-32, 179
      - IA-64, 600
      - 68000, 146
    - frame pointer (FP) (结构指针FP), 77
    - in HP3000, 604
    - in IA-32, 177
    - in IA-64,
    - pointer (SP) (指针SP), 68
    - processor (处理器)
    - push and pop operations (进栈和出栈操作), 68
    - pushdown (下推), 68
    - in 68000, 146
    - in subroutines (在子程序中), 73
  - Standards (标准)
    - IEEE floating-point (IEEE浮点), 395
    - IEEE-802, 646
    - RS-232-C, 571
    - SCI, 644
  - Start-stop format (起止方式), 566
  - State diagram (状态图), 715
  - State table (状态表), 716
  - Static memory (SRAM) (静态存储器), 297, 305
  - Status flag (状态标志)
  - Status register (状态寄存器)
  - Store-and-forward network (存储-转发网络), 634
  - Stored program (存储程序), 4
  - Strobe (选通), 241
  - Subroutine (子程序), 72
    - in ARM, 122
    - in IA-32, 177
    - in IA-64, 600
    - linkage (链接), 72
    - nesting (嵌套), 73, 78
    - parameter passing (参数传递), 74
    - in 68000, 146
  - Subtraction (减法), 29
    - floating-point (浮点), 398
  - Sum-of-products form (积之和形式), 667
  - Sun Microsystems processors
    - microSPARC, 595
    - SPARC, 594
    - UltraSPARC I, II, III, 493, 595
      - visual instruction set (VIS) (可视化指令集), 595
  - Supercomputer (巨型计算机), 2, 618
  - Superscalar processor (超标量处理器), 15, 481
  - Supervisor mode (state) (管态模式), 215, 221, 343
  - Symbol table (符号表), 63
  - Symmetric multiprocessor (对称式多处理器), 636
  - Synchronous DRAM (SDRAM) (同步DRAM), 302
  - Synchronous sequential circuit (同步时序电路), 718
  - Synchronous transmission (同步传输), 568
  - Syntax (语法)
  - System Performance Evaluation Corporation (SPEC) (系统性能评估协会), 18
  - System software (系统软件), 10
  - System space (系统空间), 343
  - System-on-a-chip (片上系统), 546
- ## T
- Testability (可测试性), 546
  - Test-and-Set instruction (测试-置位指令), 640
  - Text editor (文本编辑器), 11
  - TFT (Thin-film transistor) display (薄膜晶体管显示器), 560
  - Thread (线程), 650
  - Three-state (三态)
  - Threshold (阈值), 678, 684

Tightly-coupled multiprocessor (紧密耦合多处理器), 645  
 Time slicing (时间片), 221  
 Timers (定时器), 524  
 Timing signals (时序信号), 6  
 Token-ring network (令牌环网), 647  
 Torus network (圆环形网络), 630  
 Touchpad (触摸垫), 557  
 Trace exception (跟踪异常), 220  
 Trackball (跟踪球), 556  
 Transition time (转换时间), 686  
 Transmission (传输)  
     asynchronous (异步), 566  
     simplex and duplex (单工和双工), 565  
     synchronous (同步), 568  
 Trap (陷阱), 220  
 Tree network (树状网络), 630  
 Tri-state (三态), 415, 687  
 Truth table (真值表), 662  
 2's-complement representation (补码表示), 27  
 Two-wire link (双线连接), 564

## U

UART (Universal Asynchronous Receiver Transmitter)  
 (通用异步收发器), 259  
 UltraSPARC II, 493, 595  
 UMA multiprocessors (UMA多处理器), 622  
 User mode (state) (用户模式/状态), 215, 221, 343

User space (用户空间), 343

## V

Very large-scale integration (VLSI) (超大规模集成), 9, 20  
 Video display (视频显示器), 558  
 Virtual address (虚拟地址), 294, 338  
     address translation (地址转换), 339  
     page (页), 339  
     page fault (页故障), 342  
     page frame (页帧), 339  
     page table (页表), 339  
     translation buffer (TLB) (转换缓冲区), 341  
     virtual address (虚拟地址), 294, 338  
 Von Neumann architecture (冯·诺依曼体系结构), 19

## W

Wait loop (等待循环), 66  
 Wide area networks (WAN) (广域网), 646  
 Winchester (温切斯特), 345  
 Word (字), 5  
 Word alignment (字对齐), 36  
 Word length (字长), 5, 33  
 Workstation (工作站), 2  
 Wormhole routing (虫孔路由), 634  
 Write buffer (写缓冲区), 335  
 Write-back protocol (写回协议), 316  
 Write-through protocol (直接写协议), 316, 642



## 计算机科学丛书

- 计算机文化 (第4版) Parsons / 龚波/50.00
- 计算机科学导论 Forouzan / 刘艺/30.00
- 离散数学及其应用 (第4版) Rosen / 袁崇义/75.00
- 组合数学 (第3版) Brualdi / 冯舜玺/38.00
- 程序设计实践 Kernighan / 裘宗燕/20.00
- 程序设计语言概念和结构 (第2版) Sethi / 裘宗燕/45.00
- 计算机程序的构造和解释 Abelson / 裘宗燕/45.00
- 高效程序的奥秘 Warren / 冯速/28.00
- 编码的奥秘 Petzold / 伍卫国/24.00
- 零缺陷程序设计 Staveland / 夏昕/25.00
- 语言设计语言原理 Sebesta / 张勤/49.00
- C#程序设计 Petzold / 杨涛/30.00
- C语言解析教程 (第4版) Pohl / 麻志毅/48.00
- C程序设计教程 Deitel / 薛万鹏/33.00
- C程序设计语言 (第2版·新版) Kernighan / 徐宝文/30.00
- C程序设计语言 (第2版·新版) 习题解答 Tondo / 杨涛/15.00
- C语言参考手册 Harbison / 邱仲潘/39.00
- C语言接口与实现 Hanson / 傅蓉/35.00
- C++程序设计语言 (特别版) Stroustrup / 裘宗燕/78.00
- 《C++程序设计语言》题解 Vandevoorde / 裘宗燕/23.00
- C++面向对象开发 (第2版) Lee / 麻志毅/45.00
- C++编程思想 (第2版) Eckel / 刘宗田/59.00
- C++精髓: 软件工程方法 Sherm / 李师贤/85.00
- C++语言的设计和演化 Stroustrup / 裘宗燕/48.00
- C++程序设计教程 Deitel / 薛万鹏/22.00
- C++精粹 Pohl / 王树武/25.00
- Java语言导学 (第3版) Campione / 马朝晖/59.00

- Java编程思想 (第2版) Eckel / 侯捷/99.00
- Java语言程序设计 (第3版) Liang / 王钱/65.00
- Java程序设计教程 (第3版) 上册 Deitel / 袁兆山/55.00 (附光盘)
- Java程序设计教程 (第3版) 下册 Deitel / 袁兆山/69.00 (附光盘)
- Visual Basic.NET 程序设计专家指南 (第2版) Deitel / 龚波/118.00
- 面向对象程序设计——Java语言描述 Kalin / 孙艳春/55.00 (附光盘)
- 面向对象程序设计——C++语言描述 Johnsonbaugh / 谢君英/48.00
- 标准C++与面向对象程序设计 Wang / 李健/39.00
- Java面向对象程序设计教程 Kafura / 袁晓华/49.00
- 面向对象程序设计——图形应用实例 Laszlo / 何玉洁/35.00
- 面向对象与传统软件工程 (第5版) Schach / 韩松/48.00
- 数据结构、算法与应用——C++语言描述 Sahni / 王广芳/49.00
- 数据结构与算法分析——C++语言描述 (第2版) Weiss / 冯舜玺/35.00
- 数据结构与算法 (Java语言版) Drozdek / 周翔/49.50
- 数据结构与STL Collins / 周翔/49.00
- 程序设计语言的形式语义 Winskel / 宋国新/32.00
- 编译原理及实践 Louden / 冯博琴/39.00
- 编译原理 Aho / 李建中/55.00
- 现代体系结构的优化编译器 Allen / 张兆庆/69.00
- Linux操作系统内核实习 Nutt / 陆丽娜/29.00 (附光盘)
- UNIX操作系统设计 Bach / 陈葆钰/35.00
- UNIX环境高级编程 Stevens / 尤晋元/55.00
- UNIX编程环境 Kernighan / 陈向群/24.00
- 现代操作系统 Tanenbaum / 陈向群/40.00
- 操作系统: 现代观点 (第2版·实验更新版) Nutt / 罗宇/49.00
- 并行计算机体系结构 (第2版) Culler / 李晓明/78.00
- 结构化计算机组成 Tanenbaum / 刘卫东/46.00
- 可扩展并行计算: 技术、结构与编程 Hwang / 陆鑫达/49.00
- 并行程序设计 Wilkinson / 陆鑫达/43.00
- 并行算法导论 Xavier / 张云泉/35.00

数据库系统概念 (第4版)	Silberschatz/杨冬青/69.00
数据库原理、编程与性能 (第2版)	O'Neil/周傲英/55.00
数据库设计	Stephens/何玉洁/35.00
数据库设计教程	Connolly/何玉洁/35.00
数据库系统导论	Date/孟小峰/66.00
数据库系统实现	Garcia-Molina/杨冬青/45.00
数据库系统基础教程	Ullman/岳丽华/32.00
数据库系统全书	Garcia-Moline/岳丽华/65.00
数据挖掘: 概念与技术	Han/范明/39.00
数据仓库 (第3版)	Inmon/黄厚宽/29.00
事务处理: 概念与技术	Gray/孟小峰/96.00
信息系统原理	Reynolds/张靖/42.00
系统分析与设计教程 (第5版)	Shelly/李芳/55.00 (附光盘)
系统分析与设计方法	Whitten/肖刚/69.00
系统分析与设计方法 (第5版)	Whitten/肖刚/69.00
计算机信息处理	Mandell/尤晓东/38.00
神经网络设计	Hagan/戴葵/49.00
神经网络原理 (第2版)	Haykin/叶世伟/69.00
高性能通信网络 (第2版)	Walrand/史美林/55.00
数据广播	Tvede/徐良贤/28.00
ISDN、B-ISDN与帧中继和ATM (第4版)	Stallings/程时端/48.00
计算机网络 (第2版)	Peterson/叶新铭/49.00
计算机网络实用教程	Dean/陶华敏/65.00 (附光盘)
计算机网络实用教程实验手册	Dean/陶华敏/15.00
计算机网络与因特网	Comer/徐良贤/40.00 (附光盘)
TCP/IP详解 卷1: 协议	Stevens/范建华/45.00
TCP/IP详解 卷2: 实现	Wright/陆雪莹/78.00
TCP/IP详解 卷3: TCP事务协议、HTTP、NNTP和UNIX域协议	Stevens/胡谷雨/35.00
数据通信与网络 (第2版)	Forouzan/吴时霖/68.00
数据通信、计算机网络与开放系统	Halsall/吴时霖/69.00
数据通信与网络教程	Shay/高传善/40.00
最新网络技术基础	Palmer/严伟/20.00
Internet技术基础	Comer/袁兆山/18.00

信息安全工程	Anderson/蒋佳/49.00
密码学导引	Garrett/吴世忠/39.00
分布式算法	Lynch/舒继武/59.00
分布式系统概念与设计 (第3版)	Coulouris/金蓓弘/59.00
分布式操作系统: 原理与实践	Galli/徐良贤/38.00
分布式计算的安全原理	Bruce/李如豹/35.00
分布式系统设计	Wu/高传善/30.00
软件工程 (第6版)	Sommerville/程成/49.00
软件工程: 实践者的研究方法 (第5版)	Pressman/梅宏/59.00
设计模式: 可复用面向对象软件的基础	Gamma/吕建/35.00
软件工程: Java语言实现	Schach/袁兆山/38.00
面向使用的软件设计	Constantine/刘正捷/49.00
计算机图形学原理及实践	Foley/唐泽圣/95.00
——C语言描述 (第2版)	
计算机图形学导论	Foley/董士海/45.00
计算机图形学的算法基础 (第2版)	Rogers/石教英/55.00
专家系统原理与编程	Giarratano/印鉴/49.00 (附光盘)
模式分类	Duda/李宏东/59.00
人工智能	Nilsson/郑扣根/30.00
人工智能: 复杂问题求解的结构和策略	Luger/史忠植/65.00
机器学习	Mitchell/曾华军/35.00
数字逻辑: 应用与设计	Yarbrough/李书浩/49.00
嵌入式计算系统设计原理	Wolf/孙玉芳/65.00 (附光盘)
计算理论导引	Sipser/张立昂/30.00
人本界面——交互式系统设计	Raskin/史元春/28.00
实时系统与编程语言 (第3版)	Burns/王振宇/59.00

## 经典原版书库

计算机文化 (第4版)	Parsons/55.00
程序设计语言原理 (第5版)	Sebesta/45.00
程序设计语言 概念和结构 (第2版)	Sethi/39.00
程序设计实践	Kernighan/22.00

C++编程思想 (第2版)	Eckel/58.00 (附光盘)	高速网络与因特网——性能与服务质量 (第2版)	Stallings/45.00
C++编程思想 第2卷: 实用编程技术	Eckel/49.00	TCP/IP详解 卷1: 协议	Stevens/45.00
C++语言的设计和演化	Stroustrup/38.00	TCP/IP详解 卷2: 实现	Wright/69.00
标准C++面向对象程序设计 (第2版)	Wang/55.00	TCP/IP详解 卷3: TCP 事务协议、HTTP、NNTP和UNIX域协议	Stevens/28.00
Java编程思想 (第3版)	Eckel/66.00 (附光盘)	ISDN、B-ISDN以及帧中继和ATM (第4版)	Stallings/35.00
Java教程 (第2版)	Garside/55.00	网络互连: 网桥、路由器、交换机和互连协议 (第2版)	Perlman/36.00
Java语言导学 (第3版)	Campione/49.00	高性能通信网络	Walrand/64.00
C语言教程 (第4版)	Kelley/65.00	通信网络基础	Walrand/32.00
C语言的科学和艺术	Roberts/60.00	数据通信与网络	Forouzan/59.00
数据结构算法与应用——C++语言描述	Sahni/49.00	计算机安全 (第3版)	Pfleeger/69.00
数据结构与STL	Collins/69.00	面向对象软件构造 (第2版)	Meyer/78.00
编译原理与实践	Louden/58.00	面向对象与经典软件工程 (第5版)	Schach/59.00
高级编译器设计与实现	Muchnick/80.00	面向对象方法: 原理与实践 (第3版)	Graham/79.00
UNIX操作系统教程	Sarwar/49.00	面向对象软件开发生理 (第2版)	Elliens/59.00 (附光盘)
UNIX环境高级编程	Stevens/49.00	IT项目管理 (第2版)	Schwalbe/65.00
Linux操作系统内核实习	Nutt/32.00	Software for Use	Constantine/39.00
现代操作系统 (第2版)	Tanenbaum/48.00	编写有效用例	Cockburn/25.00
计算机体系结构: 量化研究方法 (第3版)	Hennessy/99.00	设计模式: 可复用面向对象软件的基础	Gamma/38.00
计算机组成 (第5版)	Hamacher/48.00	软件工程: Java语言实现	Schach/51.00
结构化计算机组成 (第4版)	Tanenbaum/38.00	软件工程: 实践者的研究方法 (第4版)	Pressman/68.00
并行计算机体系结构	Culler/88.00	软件工程 (第6版)	Sommerville/69.00
并行计算导论 (第2版)	Gramma/68.00	快速软件开发	McConnell/58.00
计算机体系结构: 量化研究方法	Hennessy/88.00	软件过程改进	Zahran/49.00
计算机组织与设计: 硬件/软件接口	Patterson/58.00	软件可靠性工程	Musa/49.00
可扩展并行计算: 技术、结构与编程	Hwang/69.00	软件需求管理: 用例方法 (第2版)	Leffingwell/55.00
高级计算机体系结构	Hwang/59.00	系统分析与设计	Satzinger/39.00
分布式系统概念与设计 (第3版)	Coulouris/75.00	系统分析与设计教程 (第5版)	Shelly/58.00
数据库系统导论 (第7版)	Date/65.00	信息系统原理 (第6版)	Stair/79.00
数据库系统实现	Garcia-Molina/42.00	信息论、编码与密码学	Bose/30.00
数据库系统概念	Silberschatz/65.00	现代信息检索	Baeza-Yates/49.00
数据挖掘	Witten/40.00	电子商务 (第4版)	Schneider/59.00
Internet技术基础 (第3版)	Cornet/23.00	计算机图形学原理及实践	Foley/88.00
计算机网络	Peterson/65.00		
数据通信与网络教程 (第3版)	Shay/70.00		



——C语言描述 (第2版)			
计算机图形学的算法基础 (第2版)	Rogers/45.00	组合数学 (第3版)	Brualdi/35.00
计算机图形学导论	Foley/59.00	组合数学教程 (第2版)	van Lint/58.00
机器视觉	Jain/59.00	离散数学及其应用 (第5版)	Rosen/79.00
专家系统原理与编程	Giarratano/59.00	离散数学及其应用 (第4版)	Rosen/59.00
模式分类 (第2版)	Duda/69.00	高等微积分	Fitzpatrick/69.00
模式识别 (第2版)	Theodoridis/69.00	金融数学	Stampfli/35.00
机器学习	Mitchell/58.00	数值分析 (第3版)	Kincaid/75.00
神经计算原理	Ham/69.00	概率论及其在投资、保险、工程中的应用	Bean/55.00
神经网络设计	Hagan/69.00	代数	Isaacs/65.00
人工智能	Nilsson/38.00	傅里叶分析与小波分析导论	Pinsky/49.00
知识表示	Sowa/69.00	时间序列分析的小波方法	Percival/58.00
人工智能: 复杂问题求解的结构和策略 (第4版)	Luger/69.00	数学建模 (第3版)	Giordano/59.00 (附光盘)
计算理论导论	Sipser/39.00	微分方程与边界值问题 (第5版)	Zill/69.00
人本界面——设计交互式系统的最新指示	Raskin/28.00	应用回归分析和其他多元方法	Kleinbaum/88.00
VHDL设计、表示和综合 (第2版)	Armstrong/69.00 (附光盘)	概率统计	Stone/89.00
电磁场与电磁波	Guru/68.00	多元数据分析	Latini/69.00 (附光盘)
数字逻辑应用与设计	Yarbrough/69.00	数理统计和数据分析	Rice/78.00
数字逻辑基础与Verilog	Brown/79.00 (附光盘)	随机过程导论	Kao/49.00
逻辑设计基础 (第5版)	Roth/68.00 (附光盘)	预测与时间序列 (第3版)	Bowerman/89.00
DSP算法、应用与设计	Bateman/79.00 (附光盘)	线性代数	Jain/49.00 (附光盘)
CMOS电路设计、布局与仿真	Baker/89.00	复变函数及应用 (第7版)	Brown/42.00
VLSI数字信号处理系统: 设计与实现	Parhi/79.00	实分析与复分析 (第3版)	Rudin/39.00
现代数字信号处理	Cristi/40.00	数学分析原理 (第3版)	Rudin/35.00
微机接口技术实验教程	Derenzo/59.00	泛函分析 (第2版)	Rudin/42.00
信号处理的小波导引 (第2版)	Mallat/65.00	复分析 (第3版)	Ahlfors/35.00
信号与系统的结构和解释	Edward/59.00	逼近论教程	Cheney/39.00
信号、系统和变换 (第3版)	Phillips/75.00	拓扑学 (第2版)	Munkres/59.00
数字通信导论 (第2版)	Ziemer/89.00	初等数论及其应用 (第4版)	Rosen/59.00
数字通信 (第2版)	Ian/96.00	纯数学教程 (第10版)	Hardy/65.00
机械电子系统设计	Shetty/39.00	数理金融初步 (第2版)	Ross/29.00
系统动力学 (第4版)	Ogata/75.00	代数	Artin/59.00
流体动力学导论	Batchelor/66.00	三角级数 (第3版)	Zygmund/88.00
具体数学: 计算机科学基础 (第2版)	Graham/49.00	实分析 (第3版)	Royden/45.00
		曲线与曲面的微分几何	Carmo/49.00